National Aeronautics and
Space Administration

Intelligent Systems Division
NASA Ames Research Center

Program Model Checking

A Practitioner's Guide

Masoud Mansouri-Samani, *Perot Systems Government Services*
Peter C. Mehlitz, *Perot Systems Government Services*
Corina S. Pasareanu, *Perot Systems Government Services*
John J. Penix, *Google, Inc.*
Guillaume P. Brat, *USRA/RIACS*
Lawrence Z. Markosian, *Perot Systems Government Services*
Owen O'Malley, *Yahoo, Inc.*
Thomas T. Pressburger, *NASA Ames Research Center*
Willem C. Visser, *Seven Networks, Inc.*

January 15, 2008

Version 1.1

Contents

Figures

Preface

Program model checking is a verification technology that uses state-space exploration to evaluate large numbers of potential program executions. It can be effective at detecting critical software errors that are difficult to find through traditional testing. Program model checking provides improved coverage over testing by systematically evaluating all possible test inputs and all possible interleavings of threads in a multithreaded system.

In the real world, "all possible" can be a very big number. To address this challenge, model-checking algorithms use several classes of optimizations to reduce the time and memory requirements for analysis, as well as heuristics for meaningful analysis of partial areas of the state space. This is still not always sufficient to enable exhaustive coverage. However, even with only partial coverage of a system, the ability to control thread scheduling and environment responses while monitoring the system state offers benefits over testing for finding requirements violations.

Program model checking evolved into an active research area at the end of the 1990s. After nearly a decade of investigations and case studies, best practices for applying program model checking are now emerging from various methods for capturing properties, building special-purpose test drivers, and modifying and abstracting application code. In addition, the effect of design practice on verifiability—including model checking—is being explored. Our goal in this guidebook is to assemble, distill, and demonstrate these emerging best practices for applying program model checking. We offer it as a starting point and introduction for those who want to apply model checking to software verification and validation. The guidebook will not discuss any specific tool in great detail, but we provide references for specific tools.

Specific technical areas we will address are:

- Eliciting and formalizing critical requirements and design properties to be verified using model checking
- Using the project requirements to identify critical subsystems as the focus for model checking
- Developing models of the existing hardware and software environment, including abstracted implementations for libraries used by applications, to use as test drivers during model checking
- Redesigning system elements that are difficult to test or model check
- Using heuristics to improve the effectiveness of verification
- Measuring and monitoring state-space coverage and model fidelity
- Evaluating counterexamples and spurious errors
- Using model checking to automate test case generation

Intended Audience

The guidebook is intended for both developers and test engineers. If you are a software designer, you can use the information in this guidebook to improve the verifiability of your designs; if you are an application developer, you will learn new techniques to validate your code. Test engineers can use this guidebook to develop test drivers that can be used to support both testing and model checking.

The methods we discuss for formalizing requirements and identifying critical properties will be useful to teams developing verification goals for Software Assurance during Verification and Validation or Independent Verification and Validation. We have provided guidance on configuring model-checking options and organizing and validating model-checking results. The Design for Verification guidelines are applicable to guiding system upgrades or next-generation system development.

Acknowledgements

This guidebook was developed at NASA's Ames Research Center by the Robust Software Engineering (RSE) group, led by Dr. Michael R. Lowry. The Principal Investigator for development of the guidebook was Dr. John J. Penix, and more recently, Tom Pressburger. The authors wish to acknowledge the technical collaboration of Howard Hu and his team at NASA Johnson Space Center on the application of program model checking to SAFM; we particularly thank Tanya Lippincott of the General Dynamics AIS Division for contributing her detailed knowledge of the SAFM implementation and test environment. We also acknowledge the contributions of Murali Rangarajan and his team at Honeywell to the DEOS collaboration.

Funding for development of this guidebook was provided by the Software Assurance Research Program (SARP) of NASA's Office of Safety and Mission Assurance (OSMA), and was managed by the NASA Independent Verification and Validation (IV&V) Facility.

NASA's Office of Aerospace Technology (OAT) supported the RSE group's underlying research on model checking, via NASA's Information Technology Strategic Research (ITSR) Project; Communication, Information, and Computation Technology (CICT) Program; and Engineering for Complex Systems (ECS) Program. The Constellation Program's Exploration Technology Development Program (ETDP) also supported the underlying research.

Gerard Holzmann and Michael Lowry provided feedback on early drafts and Sue Blumenberg provided editing support.

1 Introduction

1.1 What is Model Checking?

Model checking is a collection of techniques for analyzing an abstract representation of a system to determine the validity of one or more properties of interest. More specifically, it has been defined as *an algorithmic formal verification technique for finite-state, concurrent systems* (Clarke, Emerson, and Sistla 1986; Queille and Sifakis 1982; NASA 2004).

A concurrent system can be modeled by a finite-state machine represented as a directed graph consisting of:

- *nodes*, which represent the states of the system, and
- *edges*, which represent transitions between states

In theory, a model checker can exhaustively search the entire state space of the system and verify whether certain properties are satisfied. A *property* is a predicate on a system state or states, and is usually expressed as a logical specification such as a propositional temporal logic formula. If the system satisfies the property, the model checker typically generates a confirmation response; otherwise, it produces a *trace* (also called a *counterexample*) which shows the violation of the property and the major events leading to that violation. Therefore, the model checker can be used both to prove correctness of the system behavior and to find bugs in the system.

1.2 Hardware *vs*. Software Model Checking

Model checking can be applied to both hardware and software. In fact, at the system level it can be applied to both at the same time. The Intel Pentium bug in 1994 was the "disaster" that inspired the hardware industry to pursue formal verification of hardware designs by trying new techniques such as model checking, with the goal of preventing such expensive mistakes in the future. Since that time, model checking has been used extensively in the industry for hardware verification.

Model checking can also be applied to software systems at different stages of the development lifecycle. Used early in the lifecycle, it can analyze software requirements specifications (e.g., Holzmann 1990; Atlee and Gannon 1993; Chan *et al.* 1998; Chan *et al.* 1999; Heitmeyer *et al.* 1998) and software design models (e.g., Allen, Garlan, and Ivers 1998; Dang and Kemmerer 1999; Holzmann 1997). This can work well because these models can be both interesting and small enough to avoid model checking's performance limitations. Early use of model checking is effective since many safety-critical software defects are introduced early in the lifecycle (Lutz *et al.* 1998) and are expensive to correct later.

While early lifecycle analysis has its clear benefits, some software errors cannot be discovered in the requirements and design stages. Sometimes the details of the system are not sufficiently elaborated to reveal problems until implementation. Also, many errors are introduced during implementation.

NASA has encountered a number of software problems that were traced to implementation issues. For example, in 1999 the Mars Polar Lander (MPL) was lost during its final descent to the Martian surface at an estimated cost of $165 million. The most likely cause of failure was identified as a software-related problem—a single bad line of code. A variable that was not re-initialized after a spurious sensor signal associated with the craft's legs falsely indicated that the craft had touched down when in fact it was some 130 feet above the surface. This caused the descent engines to shut down prematurely; MPL was destroyed in the subsequent impact.

Such errors result from software becoming a pervasive component of aerospace systems. There are similar trends in other industries, such as the growing use of Integrated Modular Avionics (IMA) in civil aviation, which allows applications of varying criticality levels to execute on a shared computing platform. The increased scope and complexity of software naturally make it more difficult to design and validate. This suggests that changes need to be made in how aerospace systems are verified and validated (Rushby 1999).

In contrast to hardware model checking, the use of model checking for program verification has been restricted mostly to the research community. There are good reasons why this has been the case, and we will discuss these in this guidebook.

1.3 What is Program Model Checking?

Program model checking refers to the application of model-checking techniques to software systems, and in particular to the final implementation where the code itself is the target of the analysis (Figure 1). It can be effective at uncovering critical software defects that are hard to find using approaches such as traditional testing.

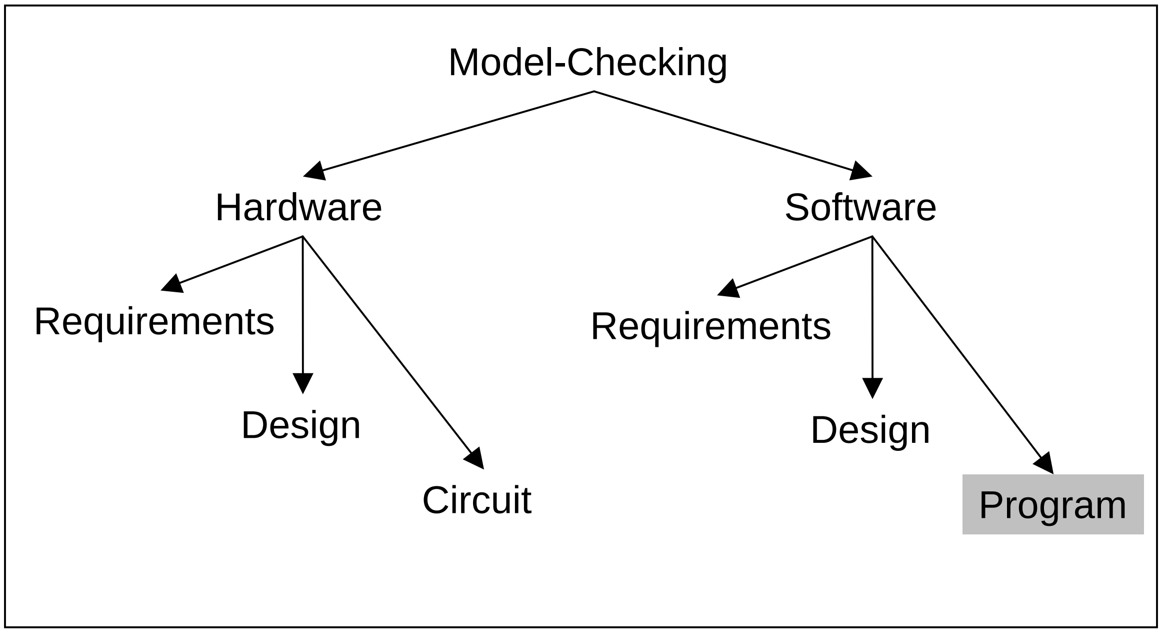


Figure 1 Model Checking

In the remainder of this guidebook we will use the term *model checking* to mean *program model checking*, unless otherwise stated.

1.4 Advantages of Model Checking

Many tools already exist to help detect problems in source code. Testing is clearly the most widely used technique and there are a variety of tools which support test data selection, test case management, test result checking, and test coverage assessment (Bezier 1990; Kaner, Falk, and Nguyen 1993). A growing number of static analysis tools can also evaluate source code for defects without actually executing the code. The capabilities of these tools range from checking simple coding conventions to detecting issues such as non-initialized variables, null pointer dereferences, and array out-of-bounds errors (Coverity; FindBugs; GrammaTech's CodeSonar; Klocwork; Parasoft's C++Test and JTest; Polyspace; Evans 1996; Dor, Rodeh, and Sagiv 1998).

For model checking to be useful in practice, it must provide clear benefits over other techniques such as testing and static analysis, either by finding more or different types of errors or by reducing cost.

One area where model checking shows benefits is in concurrent (multithreaded) systems. Testing concurrent systems is problematic because the tester has little control over thread scheduling, and concurrency errors are notoriously irreproducible (Yang, Souter, and Pollock 1998; Hwang, Tai, and Hunag, 1995). Static analysis has had better success dealing with concurrency, but it can be challenging to obtain accurate results (Naumovich, Avrunin, and Clarke 1999).

Model checking overcomes the two limitations of testing by providing *control over the order of thread scheduling* and *error traces to make test results repeatable*. By controlling thread scheduling, the model checker can evaluate any—and sometimes every—possible interleaving of threads in the system.

Another area where model checking can help is in the development of test cases. In addition to controlling thread scheduling, the model checker controls the test environment where the test data originates. Without any guidance, a model checker will naively try to generate *all* combinations of environmental behaviors as the closed application + environment system is checked. While in practice it is not possible to exhaust all possible input combinations, this ideal provides a comprehensive starting point to some new and interesting heuristics for generating test sets. In addition, because the tool itself generates the combinations of inputs, model-checking tools provide some nice techniques for building concise specifications of possible test data values. A model checker can provide much better coverage of the program because it can exhaustively and systematically look at all the possible behaviors rather than selected behaviors chosen (sometimes randomly) by QA personnel.

Even when using the more traditional techniques to come up with a comprehensive set of test cases independently, there still remain the issues of performance and limited system resources available to apply those test cases. Model checkers employ various techniques for backtracking and efficient storage of visited states to avoid revisiting the previously checked portions of the state space. This allows them to be much more efficient with respect to precious resources such as memory and time. It also allows them to provide a trace that records all the significant events along the execution path that lead to an erroneous condition, showing not only what the error was but also how it occurred.

Unlike other available techniques, a model checker can perform these tasks automatically for the most part, if the system is finite-state. Many interesting properties can be verified by a model checker out of the box without requiring much specific domain knowledge or expertise. (Specification of application-specific properties, of course, requires domain expertise.)

It is this exhaustive, systematic, efficient, and automated nature of model checking that makes it attractive for those involved in software verification and validation, and can offer higher levels of assurance compared to other currently available techniques.

1.5 Model-Checking Challenges

As described before, a model checker typically examines all the possible states and execution paths in a systematic and exhaustive manner in order to check if one or more properties hold. That's how it works in theory. But in practice and for realistic systems, "all the possible states and execution paths" can be such a large number that the model checker runs out of essential system resources such as memory before it can complete its task. In fact, in the worst case, the number of states increases exponentially with every variable or thread added to the system. This *state space explosion* problem has been the target of a great deal of research.

1.6 Objectives of this Guidebook

The state space explosion problem is a fundamental problem that cannot be removed—it can only be managed. One way to manage state space explosion is to abstract large or complex systems in some way. For example:

- Variables which are "irrelevant" with respect to a certain property can be removed before model checking, or
- Certain combinations of inputs or events can be ignored.

Determining what is relevant and what is not can be difficult: If you throw out too much, your model can't distinguish errors from valid runs, but if you leave in too much, the model checker may run out of memory before finding any errors.

There are other ways to manage the state space explosion problem. Among the most successful approaches are *approximation*, the use of appropriate *search strategies* (e.g., bounded and guided searches), and *partial coverage*, techniques which have opened the door to tackling large problem spaces efficiently. The decision as to which one of them—or what combination of them—to use in practice is not easy.

That is where this guidebook comes in. Our goal is to help you walk the line between model fidelity and tractability, and to learn something about your real programs.

The next chapter, "Programs, Properties and Models," covers the basics of applying model checking to programs. The remaining chapters focus on different approaches to getting useful results from model checking while not getting caught in a state space explosion. All of these chapters use real examples from real software systems to which we have applied model checking.

- "Test Drivers and Environment Models" looks at the central role that the model of the system environment plays in determining the size of the state space and validity of the results, and introduces techniques for building effective environment models.
- "Program Abstraction" describes how we can safely (and sometimes not so safely) reduce the number of states of the system under test.
- Sometimes it is easier to leave the model alone and instead use "Search and Partial Coverage" techniques to explore the model.
- In "Program Design Guidelines for Model Checking," we present some design and coding guidelines based on our experience.
- "Model Checkers" presents a few important model-checking systems.
- "Automated Testing" discusses approaches to black-box and white-box testing that combine *symbolic execution* and model checking techniques.
- The final chapter, "Case Studies," briefly describes several NASA-relevant case studies.

We have included two appendices with sample code: "An Example of a Primitive Type Abstraction in C++," and "Example Implementation of Minimal C++ PbC Infrastructure." The document concludes with a list of acronyms and abbreviations, a glossary, and bibliographic references.

2 Programs, Properties, and Models

2.1 Models

Building an accurate model of the system is a major and critical step in program model checking. You need to create models of both the application to be model checked and the environment in which it is run, including all the input data and event stimuli. In this chapter we mostly concentrate on modeling the target application. Environment modeling is covered separately in Chapter 3.

So what are the characteristics of a good model? A model must include the most relevant information needed for determining whether the program properties being investigated hold in the target application.

Parts of the system that do not affect the outcome of the property verification can be excluded from the modeling phase. The dangers of getting the model wrong are obvious. If you exclude or abstract too much, you may get false positives or false negatives. If you leave in too much, the model checker spends most of its time checking unnecessary code, wasting precious time and memory.

There are several different approaches you can take to obtaining a model of the system. They are, in increasing order of preference:

- Manual model construction
- Model extraction
- The program as the model

2.1.1 Model Construction

The initial research projects which applied model checking to real source code built models by hand based on relatively small parts of programs (Penix *et al*. 2000; Havelund, Lowry, and Penix 2001). This was a lot of work.

Manual model construction is time consuming and error prone. It is often difficult to get the models right. It requires domain expertise to know what to include and what to leave out. Chapter 5 of (Holzmann 2004) gives some hints, including avoidance of sinks, sources, filters, and counters. Counters are described in section 6.2.5 of this guidebook.

2.1.2 Model Extraction

Several program model checkers are based on automated model extraction, where the program is translated into the input notation of an existing model checker (Corbett 1998; Havelund and Pressburger 2000; Corbett *et al.* 2000; Ball *et al.* 2001; Holzmann and Smith

2002). Bandera (Corbett *et al.* 2000) translates Java programs to a number of back-end model checkers, including SPIN (Holzmann 2004), dSPIN (dSPIN website), SMV (SMV website), Bogor (Robby, Dwyer, and Hatcliff 2003), and Java PathFinder (JPF) (JPF website). Bandera also supports abstraction by transforming the Java programs to "abstract" Java programs which are then translated. JCAT (JCAT website) translates Java to SPIN and dSPIN. FeaVer/Modex (FeaVer website, Modex website) translates C code to SPIN. SLAM (see section 7.9) translates C code to Boolean programs as input for the Bebop model checker. FeaVer and SLAM incorporate abstraction methods into the translation process. FeaVer's abstraction is semi-automated, while SLAM uses predicate abstraction and abstraction refinement (Ball, Podelski, and Rajamani 2002) to automate abstraction during model checking.

2.1.3 Programs as Models

Program model checkers simply model check programs directly. They often adopt a modeling notation which is—or is close to—some popular implementation language such as C or Java (Visser *et al.* 2003; Stoller 2000; Musuvathi *et al.* 2002). The best possible scenario is when the language of the target program and the modeling notation are exactly the same. In this case, the program can be model checked almost directly without the need for constructing a separate model—the program itself is used as the model. This can reduce the cost of analysis by reducing the effort required to construct and maintain separate and explicit models. It also avoids the maintenance cost of trying to keep the models and the code consistent as the software is modified.

In general, however, it's not possible to avoid the construction of a separate model because of factors such as the mismatch between the modeling and target notations or the required level of detail in the model. It is often necessary to apply a combination of the techniques mentioned above to obtain a valid representation of the program to be analyzed. Parts of the system may be modeled manually, whereas other parts may lend themselves to direct translation.

2.2 Model Checking a Program

The operation of a model checker can be described in terms of the familiar metaphor of searching a graph (Figure 2). The nodes of the graph represent the *states* of the program and the links connecting the nodes represent *state changes* or *transitions*. *Branch points* in the graph represent choices resulting from, for example, scheduling decisions or different input values, which can produce multiple options as to the next state to be considered by the model checker.

The *state* of a program can be described as the bindings of variables and data members to values, the value of the program counter representing the instruction being executed, and the information identifying the current thread of execution.

The state of a program changes over time. The state graph represents the *state space* of the program as a whole—the collection of all possible program states and transitions.

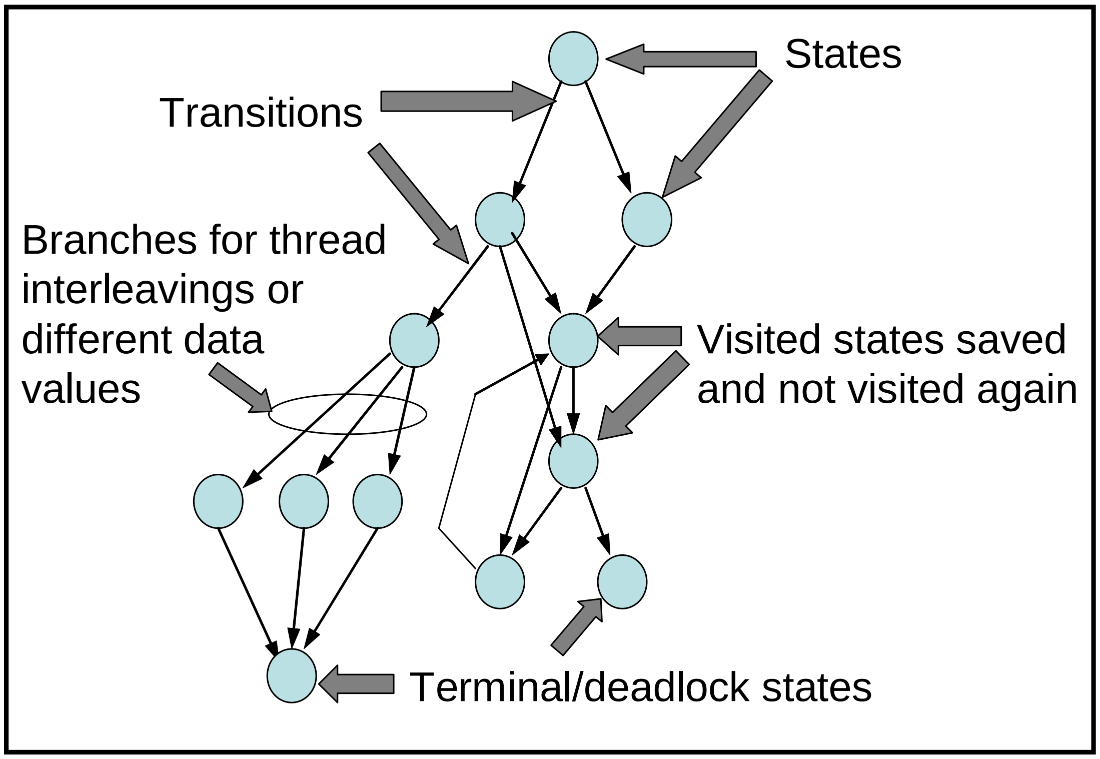

States
Transitions
Branches for thread interleavings or different data values
Visited states saved and not visited again
Terminal/deadlock states

Figure 2 State Graph

A model checker essentially tries to cover every state in the state space and follow all the transitions, covering every state reachable by any execution path.

Explicit state model checkers support a *state storage* mechanism which allows them to remember the previous states that they have visited. Once a leaf state or previously visited state has been reached, the model checker can backtrack and try the other choices that could have been taken instead. This allows the model checker to be systematic and thorough in covering the entire state space while checking the properties of interest.

2.3 Properties

A *property* is a precise condition that can be checked in a given state or across a number of states (for example, to describe behaviors in the case of temporal properties). It should specify *what* the condition must be, not *how* it is satisfied.

The overall aim of model checking is to increase the quality of verification and validation (V&V) by specifying and checking properties that cover all of the application requirements.

The best time to capture the properties is during each relevant phase of the software lifecycle (requirement, design, development, and testing) and not after the fact.

Whatever is known about the correctness and completeness of the application during each phase should be captured and preserved as additional information—ideally within, or closely associated with, the relevant artifacts—as the system is developed. This allows the designers, developers, and test personnel to be aware of all the properties and the related requirements.

Unfortunately, in practice such properties are often "reverse engineered" during the V&V process.

2.3.1 Types of Properties

Properties can be categorized along several dimensions, as described in the following subsections.

2.3.1.1 Generic vs. Application-Specific Properties

Properties can be *generic* or *application specific*. Examples of generic properties are:

- No array out-of-bounds accesses
- No divide by zeros
- No deadlocks
- No race conditions
- No null-pointer dereferences
- No uncaught exceptions

Application-specific properties are assertions about particular features or behaviors that depend on the nature of the application and typically involve domain-specific terms. For example:

- "The pitch rate ratio shall be between 0 and 1."
- "The busy flag shall be set if the input is not available for five consecutive cycles."

2.3.1.2 Safety Properties

A *safety* property asserts that nothing bad will happen during the execution of the program (e.g., no deadlocks, or no attempts to take an item from an empty buffer).

Safety properties are used mainly to ensure consistency of the program state, for example by making sure that:

- Shared resources are updated atomically (mutual exclusion), or
- The state of shared resources is kept consistent by delaying certain operations (condition synchronization)

2.3.1.3 Liveness Properties

A *liveness* property asserts that something good eventually happens, and is used mainly to ensure progress. Liveness properties prevent:

- starvation: Processes not getting the resources they need (e.g., CPU time, locks)
- dormancy: Waiting processes fail to be awakened
- premature termination: Processes killed before they are supposed to be

In general, liveness properties are harder to detect than safety properties.

2.3.1.4 Fairness Properties

A *fairness* property is essentially a liveness property which holds that something good happens infinitely often (for example, a process activated infinitely often during an application execution—each process getting a fair turn).

2.3.1.5 Temporal Properties

Properties that hold for a state in isolation are called *state* properties. *Temporal* (or *path*) properties relate state properties of distinct states in the state space to each other. Such properties are also referred to as *dynamic properties*. Temporal properties can be expressed as statements about the relation of different state properties along possible execution paths. For example, a temporal property can state that all paths which contain a send signal emitted at one state will also contain a later state where the signal has been received.

2.3.2 Identifying and Eliciting Critical Properties

Identification and elicitation of critical properties is the initial and essential step in the verification process; however, there is no well-defined and generally accepted process to follow.

For our SAFM case study in section 9.4 we used a combination of approaches in an attempt to be thorough and to understand which approaches were most useful. To do this effectively, we collaborated closely with the SAFM domain experts to identify the set of properties for model checking. Working closely with those who had intimate knowledge of the requirements, design, code, and existing testing infrastructure saved a lot of time and effort.

We then classified those properties in terms of their criticality and importance as well as their general applicability (*generic properties* such as “no divide by zeros” or “no overflows/underflows”) versus their specific relevance to SAFM (*application-specific properties*; for example, “the value of the time step shall be between the values … and …”).

Generic properties can be formulated and understood by those who do not have any application-specific domain knowledge. Model checkers such as SPIN and Java

PathFinder are able to detect violations of many such properties by default or by the user setting one or more configuration options.

In contrast, specifying application-specific properties is more challenging. As their name suggests, they often require intimate understanding of the system under analysis. Therefore this information is best formulated and captured by or in close collaboration with the original requirements engineers, designers, and developers who have such domain-specific knowledge.

Even with intimate knowledge of the system, the process of identifying the properties of interest is usually not straightforward. These properties must make sense and be the right ones to use for verification of the system. You may also need to further constrain a property or widen its applicability. They must be specified, developed, validated, and verified in the same way as any other software artifact.

In order to do this and also to get an assessment of the coverage of those properties using the existing testing framework, we encoded those properties and instrumented the code using AspectC++ (described in section 2.3.4.2) and ran the existing tests using SAFM's existing test driver. We showed the results of our findings (including violations of the supposed properties) to the customers, and through an iterative process adjusted the properties until we were satisfied that we had come up with a set of valid properties for our model-checking work.

2.3.3 Specifying Properties

Different tools use different notations for specifying properties. In this section we describe some of the techniques.

2.3.3.1 Property Specification in Java PathFinder

In Java PathFinder, properties are specified in Java. JPF can check a number of properties out of the box:

- No Deadlocks
- No Assertion Violation
- No Uncaught Exceptions (i.e., no exceptions that are not handled inside the application)
- No Race conditions

For assertion violations, you must add Java `assert` statements to the source code. Java PathFinder then reports when those assertions are not satisfied. For other types of default properties, no special annotations are necessary. All that is needed is that they are activated through configuration options.

JPF also allows you to add new properties through a well-defined interface. This interface lets you access not only the state of the program under analysis but also the state of the model checker itself.

The same interface can also be used to implement temporal properties, but it is your responsibility to write the code (e.g., an automaton) to keep track of successive states used to verify the property.

All the properties can be activated and deactivated as required through configuration options.

2.3.3.2 Property Specification in SPIN

SPIN allows specification of assertions, which check a predicate of a state at a point in an execution, and also allows for specification of progress properties (e.g., no starvation) by allowing statements to be labeled as *progress states*. A report is generated if there are infinite executions that do not pass through one of these marked states. SPIN also provides a facility for stating *never claims*, behavior that should never occur. Never claims are checked at each step in the execution. This allows you to state an invariant (which may even be a path property) that should never be violated. You can specify temporal properties using a timeline editor.

Linear Temporal Logic (LTL) is a language to express properties that hold for paths throughout the state space. The SPIN model checker allows temporal properties to be expressed in this language; the properties are then compiled into never claims. Formulas in LTL are built of state properties and the following operators: `[]` (always), <> (eventually), and `U` (until), plus the standard logical operators. The semantics is as follows:

- *P*: The property *P* holds in the current state.
- `[]`*P*: The property *P* holds in this state and in all future states.
- <>*P*: The property *P* eventually holds in this or a future state.
- *P* `U` *Q*: The property *P* holds in this state and future states until a state is reached where the property *Q* holds.

The send/receive temporal property mentioned in section 2.3.1.5 can be expressed in LTL as

```
[](Send => <> Receive)
```

where => is logical implication. Some simple properties can be elegantly expressed in LTL; however, many desirable properties become quite complex expressions.

SPIN provides the trace assertion facility to formalize statements about valid or invalid sequences of operations that processes perform on message channels. SPIN message channels are discussed in section 7.2.

2.3.3.3 Property Patterns

One important obstacle to using temporal logic is the difficulty of expressing complex properties correctly. Dwyer and his colleagues have proposed a pattern-based approach to the presentation, codification, and reuse of property specifications for finite-state verification (Dwyer, Avrunin, and Corbett 1998; Dwyer, Avrunin, and Corbett 1999) and (Spec Patterns website).

The patterns enable non-experts to read and write formal specifications for realistic systems (including reactive systems) and facilitate easy conversion of specifications between different formalisms.

The system allows patterns like "*P* is absent between *Q* and *S*" or "*S* precedes *P* between *Q* and *R*" to be easily expressed in and translated between Linear Temporal Logic (LTL) (Manna and Pnueli 1990), Computation Tree Logic (CTL) (Clarke *et al.* 1986), Quantified Regular Expressions (QRE) (Olender and Osterweil 1990), and other state-based and event-based formalisms. (Dwyer, Avrunin, and Corbett 1999) also performed a large-scale study in which specifications containing over 500 temporal properties were collected and analyzed. They noticed that over 90% of these could be classified under one of the proposed patterns.

2.3.4 Inserting Property Oracles

A *property oracle* is a program or a piece of code that says whether or not a property holds during model checking. The process of inserting this (often conditionally compiled code) in the program is called *instrumentation*.

While non-functional properties like deadlocks and memory leaks do not have to be expressed explicitly in the program sources, functional properties, describing application-specific functionality like state consistency and correctness of computations, needs explicit instrumentation.

For program model checking, this instrumentation takes the form of annotations and code extensions which are inserted in the source. You must be careful about where these are inserted. Do not insert them in the code at intermediate states where the property is not expected to hold, only where the program is in a consistent state. We call these places *checkpoints*. For example, the code for calculating some time interval value may span multiple statements. The checkpoints for verifying that the value is within some predefined range can be before the first and after the final statement involved in that calculation. Checking the code at any other point in between may lead to false positives or negatives, as the check may be using intermediate values.

You can use the requirements, the verification and validation matrix, the existing design models (if any), and the expertise of those with domain knowledge to identify these checkpoints. In the case of SAFM, for some properties we had to get confirmation from the domain experts to make sure that we were inserting our instrumentation at the right places.

There are different techniques for inserting property oracles into programs. Two of these mechanisms discussed here are Programming by Contract and Aspect-Oriented Programming. Note that these are complementary techniques, and may be used in combination.

2.3.4.1 Programming by Contract

The major mechanism for expressing functional properties is the use of *assertions*—statements involving conditions expressed as Boolean expressions which cause program termination if the condition is not met.

```
#include "assert.h"
...
assert((x > 10) && (y < 50));
```

Assertions are usually conditionally compiled, and therefore they should be programmed side-effect-free with respect to object states. For C/C++, this is implemented by means of the pre-processor, not integrated into the language. It works well for "flat" procedural programs, but has limitations with object-oriented programming (OOP), as discussed later.

The preferred way to integrate assertions into an object-oriented system is through Programming by Contract (PbC) (Meyer 1992). PbC extends and provides specialized assertions by defining standard scopes and evaluation rules for them, especially in the context of object-oriented programming. The following assertion types are part of PbC:

- *Pre-condition*: An assertion checked before a function/method body is executed.
- *Post-condition*: An assertion checked before a function/method exits, including exception control flow.
- *Loop invariant*: An assertion that is true at the beginning of the loop and after each execution of the loop body.
- *Loop variant*: An assertion that describes how the data in the loop condition is changed by the loop. It is used to check forward progress in the execution of loops to avoid infinite loops and other incorrect loop behavior.
- *Loop bounds*: An assertion that checks that the loop can only be traversed a bounded number of times.
- *Class invariant*: An assertion that is evaluated before and after each invocation of a public function and can be thought of as a way to factorize contracts.
- *Assertion*: An assertion not attached to standard language constructs.

The main purpose of contracts is to clearly define responsibilities between the supplier (the callee must fulfill post-conditions) and the consumer (the caller must fulfill pre-conditions), and hence help to avoid errors that are caused by ambiguity. Compliance is tested with executable code that can be optionally compiled into the application. A typical example is to test parameter and return values for constraints:

```
class Foo {
    //@ invariant: (v >= VMIN) && (v <= VMAX)

    double v;

    double accelerate (int t, int a) {
        //@ requires:  (t > 0) && (a > 0)
        //@ ensures:   (v > \old(v)) && (\result == v)

        v = …
        …
        return …
    }
}
```

Ideally, contracts should be inheritance aware—a call to a redefined function in a derived class should succeed if any of the pre-conditions are met (weakening), and upon return should ensure that all post-conditions hold (strengthening).

The lack of properties that can be tested and used as input for formal methods has traditionally hampered the V&V process. In many cases, these properties are created after the development has been completed, leading to additional costs and missing regression test suites during development. Incorporating contracts into the sources even before functions are implemented ensures that:

- Developers have a clear understanding of responsibilities (self-documenting sources), which is kept up to date throughout the development process.
- Testers automatically get a set of potentially fine-grained testing conditions that are especially useful for creating unit tests and regression test suites.
- External V&V does not have to "guess" functional properties later on.
- V&V does not require subsequent instrumentation of sources for each new version, which normally has to be done manually.

2.3.4.1.1 Support for Programming by Contract

Programming by Contract can be supported on several levels, depending on the programming language and available third-party tools.

Direct support by language (e.g., Eiffel, Digital Mars C++)

```
class FOO
  feature
    v : DOUBLE

    accelerate (t, a : INTEGER) : DOUBLE is
        require
            positive_input: (t > 0) and (a > 0)
        ensure
            v_changed: (v /= old v)
            result_consistent: (Result = v)
        …
        v := …

        do
            from  …
            invariant …
            variant …
            until …
            loop
            …
        end -- loop
        …
        Result := …
    end -- accelerate

  invariant
    v_constraints: (v >= VMIN) and (v <= VMAX)
end -- FOO
```

While direct support by language provides the most powerful support for contracts (e.g., loop variants/invariants), it depends on languages or language extensions that are not mainstream, and therefore might not be suitable for all projects.

The two statically typed (compiled), publicly available programming languages supporting PbC at this level are Eiffel (Eiffel Software) and D (Digital Mars). Both are contemporary object-oriented languages, implemented in proprietary and open source compilers, supporting several hardware architectures and operating systems. Eiffel provides deeper PbC support (e.g., loop variants and invariants), and loosely resembles Pascal/Ada in terms of declaration and expression syntax. D resembles C/C++, but is not yet widely used and is less mature than Eiffel. Both languages have established user and developer communities, but cannot be considered as mainstream as C++.

While a PbC proposal exists for the upcoming C++0x standard (Abrahams *et al.* 2005), it is not yet clear if and when compiler vendors will support it. The syntax resembles D, in that it requires grammar extensions:

```
template< class T, class Alloc = allocator<T> > class vector {
  static invariant {
    static is_assignable<T>::value :
            "value_type must be Assignable" ;
      ...
  }
  invariant {
    size() <= capacity(); ..
  }
  void push_back( const T& val )
    precondition  { size() < max_size(); }
    postcondition { back() == val;
                    size() == __old size() + 1;
                    capacity() >= __old capacity(); }
  ...
};
```

Embedded Languages in Magic Comments

In this approach, contracts are specified using a separate notation that is embedded in comments, using special compilers to turn annotations into instrumentation. Two typical implementations of this category are SPARKAda and the Java Modeling Language (JML).

SPARKAda (SPARKAda website) is an Ada95 subset with a rich set of annotations that exceeds runtime instrumentation. The SPARKAda tool suite includes extensive static analysis and proof checkers.

```
package odometer
--# own Trip, Total : Integer;
is
  procedure Inc;
  -- # global in out Trip, Total;
  -- # derives Trip from Trip & Total from Total;
  -- # post Trip = Trip~ + 1 and Total = Total~ + 1;
  …
end Odometer;
```

While SPARKAda is well documented, it is only available from a single commercial vendor.

The Java Modeling Language (JML website) is an open source language extension for Java. It provides additional expressions that especially focus on avoiding side effects of contracts.

```
//@ requires x >= 0.0;
//@ ensures
//@   JMLDouble.approximatelyEqualTo(x, \result * \result, eps);

public static double sqrt(double x) {
/*...*/
  }
```

Using a contract language that is embedded in comments, but can easily access features of the relevant program scope, is almost as powerful as direct language support. The downside is that this approach usually requires a target language compiler re-implementation (an Ada or Java compiler), which imposes a serious tool dependency.

Approximation with available language and pre-processor features

Since there is no language support for PbC in mainstream languages such as C++, this has to be modeled. Modeling PbC capabilities using existing language features avoids specific language and compiler issues, but can require significantly increased design and programming effort, especially when used with object-oriented programming language class hierarchies. The approach has an increased probability of introducing errors or inconsistencies in the contract implementation itself.

The major issue for pre- and post-conditions is inheritance. In order to preserve semantics, a pre-condition in an overriding method should always be equal to or weaker (accept more) than the overridden method, and a post-condition should always be equal to or stronger (ensure more) than the overridden method.

For example:

```
class A {
  ...
  #ifdef DEBUG
  bool _pre_foo (...) const
  {...}
  bool _post_foo (...)
const
  {...}
  #endif

  virtual void foo (...) {
    assert(_pre_foo(...));
    ...
    assert(_post_foo(...));
  }
};
```

```
class B : public A {
  ...
  #ifdef DEBUG
  bool _pre_foo (...) const {
    return (/* new cond */) ||
           A::_pre_foo(...);
  }

  bool _post_foo (...) const {
    return (/* new cond */) &&
           A::_post_foo(...);
  }
  #endif

  virtual void foo (...) {
    assert(_pre_foo(...));
    ...
    assert(_post_foo(...));
  }
};
```

Without language support, if you try to make assertions based on the old and new states, you must store old states explicitly, either as data members or local variables. Since this has a negative impact on the state space, use old states carefully.

Without tool support, invariants also have to be explicitly turned into pre- or post-conditions. For example:

```
class Double {
  private:
    double v; ..
    double min, max;

    bool invariant () const { return ((v>=min)&&(v<=max)); }

    Double& operator+= (const Double& d) {
      assert ((v!=NAN) && (d.v!=NAN) && invariant()); // precond
      ..
      assert ((v != NAN) && invariant()); // postcond
      return *this;
    } ...
};
```

In the absence of direct data member access, class invariants only have to be checked on exit from non-const function members.

In general, among the tasks that you need to consider are:

- Creating dedicated `invariant()` member functions for classes
- Ensuring that invariant checks are explicitly called in pre- and post-conditions
- Ensuring that post-conditions are called upon each exit point of a function (which usually is done by enforcing a single exit point—a general coding best practice)
- Ensuring that return expressions cannot violate post-conditions (for instance, with temporary objects)

Within limits, you can mitigate these efforts by creating a dedicated infrastructure of functions and macros:

```
#include "pbc.h"  // defines macros
                  // CONTRACT_X, REQUIRES, ENSURES, RETURN

double foo (double d) {
  CONTRACT_RP(
    double, foo,         // return type & function name
    double, d,           // checked parameter type(s) & name(s)
    REQUIRES( d>0),          // preconditions
    ENSURES( (d == _d) &&    // post-condition ('_d' refers
             (result< 40.0)) // to the 'd' value upon
  );                         // entry, 'result' is the
                             // return value)
  …
  if (…) {
       RETURN(bar(d));   // macro makes sure we support
  } else {               // return value postconditions
       RETURN(baz(d));   // even with multiple exit points
  }
}
```

A C++ example implementation of such an infrastructure can be found in Appendix B.

While this approach is the most pragmatic one, and works with existing compilers, it does not provide the same amount of support (especially with respect to sub-contracts in class hierarchies), and requires additional efforts to:

- ensure that none of the PbC functions or macros are redefined;
- configure desired assertion actions (error handlers);
- avoid obfuscation of "normal" control flow and stack layout (debugging); and
- provide inspection mechanisms to analyze the context of assertion violations.

You can reduce the amount of redundant information between function header and contract definition (return type, class name, function name, etc.) by using more powerful external preprocessors, like the Unix M4 preprocessor. When choosing this approach, encapsulate non-C++ macros into comments, to ensure compatibility with existing tools (compilers, development environments, etc.).

```
// PBC_FUNC(
double mod_ab(double in_a, … //
)

  PBC_PRE(in_a >=   0.0);
  PBC_PRE(in_a <= 100.0);

  PBC_POST(ret >=   0.0);
  PBC_POST(ret <= 100.0);
```

This syntax could be further improved by a specialized contract preprocessor, but such a tool is not yet available.

An alternative is to use C++ multiple inheritance and template classes to implement contracts, and to encapsulate pre- and post-conditions and invariants in functor objects, which can also be used to control contract side effects (e.g., Guerreiro 2001). This approach is safer than the use of macros, but requires considerably more C++ knowledge. It also uses C++ features which are more expensive in terms of model checking, such as multiple inheritance, templates, and extensive creation of temporary object clones.

Even though the mainstream languages lack direct support of all of the PbC concepts, many of them either directly (e.g., Java) or indirectly (e.g., C++ through macros) support at least the specification of simple assertions. We highly encourage use of these assertions, as they provide one of the basic mechanisms for model checkers to detect defects.

2.3.4.1.2 Other Uses of Programming by Contract

Contracts can also be used for dynamic, temporal properties (Mehlitz and Penix 2005), which are especially useful to verify protocol compliance (for instance, requiring the test state to keep track of previous evaluations). However, this does not provide the documenting aspect of contracts, but rather uses them as a convenient instrumentation mechanism.

There also have been attempts to use contracts as input for theorem provers (ACL, SPARKAda, Eiffel), but this usually requires additional language features and significantly more effort.

2.3.4.2 Aspect-Oriented Programming

Detection of certain program properties may involve instrumenting many entities such as classes and methods that are spread across the entire program—for example, detecting whether every connection `open()` call is followed by a connection `close()` call, or whether for all the classes that extend (or inherit from) a given base class the `initialize()` method is invoked within its constructor. A problem with detecting such properties is the amount of manual effort that is required to instrument the program. Manual instrumentation of the code to verify such properties is difficult and error-prone. Also, the instrumentation has to be done all over again for newer versions of the software.

Aspect-oriented programming (AOP) alleviates these problems by treating the instrumentation for such properties as a cross-cutting concern.

AOP languages such as AspectJ and AspectC++ support expressions that can encapsulate such concerns in special classes called *aspects*. An aspect can change the behavior of the program by applying additional behavior called *advice* at various *join points* in a program. You can specify a query that detects whether a given join point matches. Such a query is called a *pointcut*. An aspect can also make structural changes to other classes, such as adding members or parents.

For example, the following aspect specification can be used to check whether there are an equal number of calls to the `fopen()` and `fclose()` methods when the program ends. A model checker can capture the exception raised by the assertion violation when "counter > 0".

```
aspect FileOpenCloseCounter {
  int counter;
  FileOpenCloseCounter() {
    counter = 0;
  }

  advice call("% %::fopen(...)"): after() {
    ++counter;
  }

  advice call("% %::fclose(...)"): after() {
    if (counter > 0)
      --counter;
  }

  advice execution("% main(...)"): after() {
    assert (counter =< 0);
  }
};
```

Many AOP languages support method executions and field references as join points. The developer can write a pointcut to match, for example, all field-set operations on specific fields, and code to run when the field is actually set. Some languages also support extensions of specific classes with new method definitions. AOP languages vary based on the join points they expose, the language they use to specify the join points, the operations permitted at the join points, and the structural enhancements that can be expressed.

We used AOP, and in particular AspectC++, for code instrumentation in order to check many of the properties in our SAFM case study (Figure 3). We analyzed several versions of the software, and the ease of instrumentation that the AOP techniques provided saved a lot of time and effort.

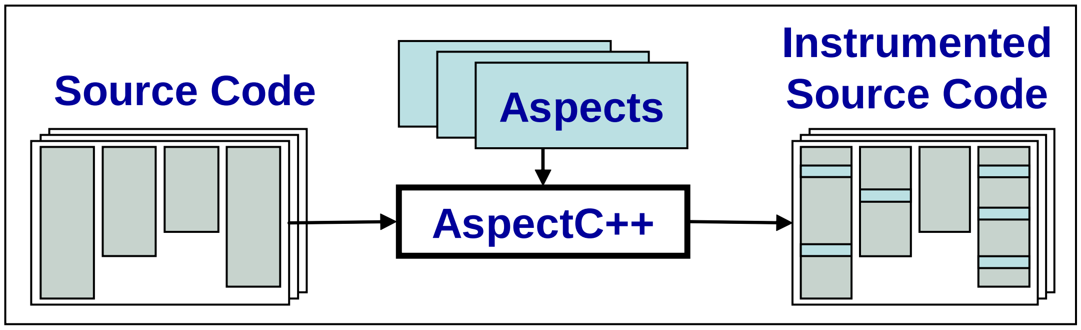


Figure 3 Code Instrumentation using AspectC++

2.3.5 Understanding Property Violations

When a property violation is detected by the model checker, the model checker typically generates a counterexample trace. The trace points to the location where the violation was detected, but quite often what is reported is only a symptom of a bug which occurred much earlier in the execution trace. At this stage, you need to localize the root cause of the property violation.

Quite often one bug may cause the code to fail in many different ways. Selection of suitable search strategies allows us to find the shortest paths to the problem, thereby minimizing the distance between the detected property violation and the root cause. Chapter 5 describes some of these strategies.

One common technique for localizing the root cause of the violation is to let the model checker generate multiple counterexamples, and use these counterexamples collectively to find the root cause.

3 Test Drivers and Environment Models

3.1 Analyzing Open Programs

A key problem in software model checking is *environment modeling*. Software model checkers such as JPF or SPIN can analyze only closed, executable programs—that is, programs that operate on fully specified inputs. However, most software systems are fundamentally open, since their behavior is dependent on patterns of invocation of system components and values defined outside the system but referenced within the system. Whether you are reasoning about the behavior of whole programs or about program components, a model of the environment is therefore essential in enabling model checking.

An environment model for a software component (*unit*) establishes an abstract runtime context in which the unit executes. It contains both *data* information (e.g., values that flow into the unit) and *control* information (e.g., a sequence of method invocations in the unit interface) that influence the unit under analysis. This model should be small enough to enable tractable verification, but sufficiently detailed to not mask property violations.

Environment models allow developers to analyze individual units early in the software lifecycle, when they become code complete, and possibly before the whole software system has been developed and integrated. In this case, model checking is used as an adjunct to *unit testing*, and the environment models play the role of a testing harness. Environment models are also used in the context of verifying large systems: The system can be broken up into smaller parts which can be verified separately and more efficiently. In this case, the environment model for an analyzed component represents the rest of the system with which the component interacts. The environment encodes only the interface behavior and is much smaller than the actual components it represents, enabling more efficient verification. Environments can also be used to model non-source-code components (for example, hardware components when analyzing embedded software) and native libraries that cannot be analyzed directly by a software model checker.

3.2 Universal Environments

Unit testing involves the definition of *drivers* and *stubs*. Drivers are program components that invoke operations on the unit under test. Stubs are program components that implement operations invoked by the unit. Stubs and drivers can be defined to also represent parallel contexts representing those portions of an application that execute in parallel with and engage in inter-thread communication with the procedures and threads of the software unit under test. In a similar way, environment models consist of a collection of drivers and stubs that together with the analyzed unit form a closed system that is amenable to model checking. The *universal environment* is capable of invoking (or refusing) any operation to or from the unit, in any order.

To construct such environments, you need a description of the classes, interfaces, and packages that make up the unit, and the unit operations that are possible. These include the method (procedure) invocations in the unit's interface, methods invoked by the unit, global variables changing their values, etc. For simplicity, in this chapter we phrase our discussion and examples in terms of the Java language. In this case, a software component can be a Java class and the operations are its public methods and the methods invoked by the class, together with the global variables such as public fields that change their value.

For example, consider the following Java class `IntSet` that implements sets of integers. The set operations consist of the public methods `isEmpty`, `add`, `remove`, and `contains`. The implementation also supports iteration over set elements (method `apply`) by invoking a user-defined callback routine (method `doWork` in interface `CallBack`) for each element in the set.

```
public Interface CallBack {
   public boolean doWork();
}
class IntSet {
public IntSet() {}
   public boolean isEmpty() {}
   public void add(int elem) {}
      public void remove(int item) {}
      public Boolean contains(int item) {}
   public void apply(CallBack cb) {}
}
```

The universal driver for this class is a thread that invokes any set operation in any order. You can write multiple threads executing the driver code to exercise the unit in a concurrent environment. The stub for procedure `doWork` is implemented in class `CallBackStub`.

```
public class UniversalSetDriver extends java.lang.Thread {
   public void run() {
      IntSet s0 = new IntSet();
      CallBack cb0 = new CallBackStub();

      while(chooseBool())
      switch(chooseInt(5)){
         case 0: s0.isEmpty(); break;
         case 1: s0.add(chooseInt(10)); break;
         case 2: s0.remove(); break;
         case 3: s0.contains(chooseInt(10)); break;
         case 4: s0.apply(cb0);
      }
    }
  }

  public class CallBackStub implements CallBack {
    public Boolean doWork() { return chooseBool(); }
  }
```

These environments use *modeling primitives* that are interpreted by the underlying model checker as a non-deterministic choice over a set of values. For example, `chooseBool()` represents a choice between true and false and `chooseInt(n)` a choice over 0 .. n-1. Note that we assumed here that the parameters in the procedure calls take values from a small finite domain (1 … 10). This ensures that the sets created by the universal environment cannot grow without a bound, and therefore the unit can be analyzed thoroughly with a model checker. This is not always the case, and abstraction often must be used in conjunction with environment generation to ensure that the local data of the unit and the data that flow between the unit and the environment range over finite domains (see Chapter 4).

3.3 Filter-Based Methods

After the universal environments are generated, they are combined with the code of the unit and then a model checker such as JPF can be used to verify unit properties (e.g., that there are no runtime exceptions). In many cases, behavioral information about unit interfaces, rather than just signatures, can be exploited to refine the definition of the environment used to complete the unit's definition. For example, calling `remove` on an empty set raises a runtime exception. However, the developer of `IntSet` may know that the set is always used in a context in which `remove` is called only on a non-empty set. In this case, he or she may want to encode this assumption about the calling context directly into the environment model:

```
  public class RefinedSetDriver extends java.lang.Thread {
    public void run() {
      IntSet s0 = new IntSet();
      CallBack cb0 = new CallBackStub();

      while(chooseBool())
      switch(chooseInt(5)){
        case 0: s0.isEmpty(); break;
        case 1: s0.add(chooseInt(10)); break;
        case 2: assume(!s0.empty()); s0.remove(); break;
        case 3: s0.contains(chooseInt(10)); break;
        case 4: s0.apply(cb0);
      }
    }
  }
```

Here, the modeling primitive `assume(cond)` is used to instruct the underlying model checker to backtrack if `cond` is not true (in JPF: `Verify.ignoreIf(!cond)`). Checking the set implementation with this refined environment will yield no runtime exceptions.

3.3.1 Discharging Assumptions

Environment assumptions (or filters) capture the knowledge about the usage of software units and they help to refine a naively generated environment. These assumptions need to be subsequently checked against implementations of the missing components (that represent the environment) in order to make sure that the entire system satisfies the desired properties. For example, you need to check `assert(!empty())` whenever `remove` is invoked by a client application that uses `IntSet`.

Assumptions can be encoded in a variety of forms, such as predicates `(!empty()`, in the set example), finite-state machines, temporal logical formulas, or regular expressions that encode patterns of component operations that the environment may execute.

Checking component property (P) under environment assumptions (A) forms the *assume guarantee paradigm*, and it is the basis of many formal frameworks for modular verification (Jones 1983; Pnueli 1984): P is the guarantee that the component provides under assumption A. When A and P are temporal logical formulas, then it is possible to model check directly $A => P$ on the unit closed with the universal environment; here "=>" denotes logical implication: P is checked only on the behaviors that satisfy (i.e., are filtered by) A. Alternatively, as discussed for the set example, you can encode the assumptions directly into the environment (in which case A is eliminated from the formula to be model checked).

Regular expressions are a familiar formal notation to many developers, and often easier to use than temporal logics for specifying environment assumptions. Regular expressions defined over the alphabet of unit operations describe a language of operations that can be initiated by the environment. As an example, `java.util.Iterator` presents a standard interface for generating the elements stored in a Java container such as a set. This interface assumes that for each instance `i` of a class implementing the Iterator, all client applications will call methods in an order that is consistent with the following regular expression assumption:

```
    (i.hasNext; i.next(); i.remove()?)*
```

This expression encodes sequencing (a call to `hasNext()` must precede a call to `next()`, otherwise a runtime exception may be thrown if the iteration has no more elements) and also optional calls `(remove())` that can be repeated 0 or more times (`*`). This assumption can be easily translated into code to encode a driver or the Iterator, or into an LTL formula that can then be used to be discharged with SPIN.

3.4 Tool Support and Applications

Environment models are essential for software model checking. The problem of environment generation is difficult and it is the subject of active research. There is little tool support publicly available, most of what is available comes from academia or research labs. One such tool, BEG (BEG website) from Kansas State University, is described in more detail below.

The Bandera Environment Generator (BEG) tool builds environment models for Java components. It uses static analysis to discover the interface of the analyzed component and it uses filters (regular expressions and temporal logic formulas) and available code from the rest of the components to build precise environment models. BEG can be used in conjunction with the Bandera tool set or JPF for model checking. BEG has been applied to Java applications developed at NASA Ames and Fujitsu.

While BEG proved to be quite useful in generating environments for small systems, the tool support is much more valuable when applied to larger software systems. One such application is described in (Tkachuk, Dwyer, and Pasareanu 2003) where BEG was applied to Autopilot, a Swing-based GUI for an MD-11 autopilot simulator used at NASA. The application code consists of more than 3600 lines of code clustered in one class. The system makes intensive use of `java.awt` and `java.swing` GUI frameworks that influence the behavior of the system; the main thread of control is owned by the framework and application methods are invoked as application callbacks. To analyze the system, BEG was used to generate stubs for all the GUI framework components and to generate drivers that encode assumptions about user behavior. JPF was used to check for mode confusion properties. JPF detected a mismatch between the user model and the software's state (encoded as an assertion violation), indicating potential mode confusion. As mentioned in (Tkachuk, Dwyer, and Pasareanu 2003), a previous effort to build an environment for this application required several months of manual work and yielded an environment model that was inconsistent with the actual environment implementation. From relatively simple assumption specifications, BEG generated an environment in less than 4 minutes that was consistent with the implementation.

4 Program Abstraction

4.1 Introduction

Abstraction is essential for software verification. Without abstraction, a realistic software application is usually too large to be analyzed exhaustively with a model checker. Abstraction aims to transform a program into another program that still has some key properties of the original program, but is much simpler, and therefore easier to analyze. In model checking, abstractions are used to reduce the size of a program's state space in an attempt to overcome the memory limitations of model-checking algorithms (Cousot and Cousot 1997; Cousot and Cousot 1999; Ball *et al.* 2001; Henzinger *et al.* 2002; Havelund and Shankar 1996; Clarke, Grumberg, and Long 1994; Saïdi 1999).

Given a program and a property, the strategy of model checking by abstraction can be summarized as follows.

1. Define an abstraction mapping between the concrete program and an abstract program.
2. Use the abstraction mapping to transform the concrete program into an abstract program; usually the property needs also to be transformed into an abstract property.
3. Apply model checking on the abstract program.
4. Map the results of model checking the abstract program back to the original program.

We distinguish between *data abstractions*, which replace the large domains of program variables with smaller domains, and *control abstractions*, such as slicing, which remove program components that are irrelevant to the property under analysis. This chapter is mainly concerned with data abstraction. Abstractions can be further characterized by the way they preserve a property or class of properties being verified, or by the way they approximate the behavior of the system being verified.

4.1.1 Property Preservation

To use abstraction to show that a property holds on a concrete program, any abstractions must be property preserving. *Property preservation* enables you to take the results of checking the property on the abstracted program and to map them back to the original program. There are several forms of property preservation.

4.1.1.1 Weak Preservation

An abstraction of a concrete system is *weakly preserving* if a set of properties true in the abstract system has corresponding properties in the concrete system that are also true.

4.1.1.2 Strong Preservation

An abstraction is *strongly preserving* if a set of properties with truth values of either true or false in the abstract system has corresponding properties in the concrete system with the same truth values. Strong preservation may not seem to allow much room for simplifying the system during abstraction. However, because property preservation is defined with respect to a specific property (or a specific class of properties, as in LTL), there are many useful strongly preserving abstractions.

Weakly preserving abstractions can be much more aggressive in reducing the state space of a program and therefore they enable more efficient verification. The price paid is that the reported property violations may not correspond to real violations in the concrete, unabstracted program.

4.1.1.3 Error Preservation

When verification is used to disprove properties (find bugs), a third type of property preservation comes into play: An abstraction is *error preserving* if a set of properties that is false in the abstract system has corresponding properties in the concrete system that are also false.

4.1.2 Behavior Approximation

Abstractions are also described in term of the relative behavior of the abstract system versus the concrete system. A program's "behavior" is defined as the set of possible execution paths, also called *traces* or *computations*. These are the allowable sequences of program states, starting from a valid initial state.

4.1.2.1 Over-approximation

Over-approximation occurs when there are more behaviors in the abstract system than the concrete system. This approach provides a popular class of weakly preserving abstractions for properties that hold on all paths of a program (e.g., safety properties or more general universally quantified path properties such as LTL). Weak preservation holds because when behaviors (execution paths) are added to create the abstract system, any property that is true for all paths is true for any subset of those paths—including the subset that exactly describes the behavior of the concrete system. However, due to over-approximation, errors that are reported for the behaviors of the abstract program may not correspond to behaviors in the original program. To enable verification, these "spurious" errors must be removed by reducing the degree of over-approximation. The goal is to find an abstraction that is precise enough to allow verification of a property, while abstract enough to make verification practically tractable.

4.1.2.2 Under-approximation

Under-approximation occurs when behaviors are removed to create the abstract system. This approach corresponds to error-preserving abstractions (where the properties are safety or more general universal properties such as LTL). As mentioned, these abstractions are most useful for finding whether properties are false; hence, they can be used for debugging. Simple examples of under-approximation include: limiting the size of the data domains of a program (e.g., reduce the size of a communication channel) or using a model checker with a limited search depth or until it runs out of resources (memory or time). Program testing is another example: A test driver uses test cases to lead the system through a *subset* of the possible program executions. Testing is an error-preserving abstraction for these properties.

4.1.2.3 Exact Abstraction

An abstraction is *exact* (with respect to a property or a class of properties) if there is no loss of information due to abstraction. Exact abstractions are strongly property preserving. Under-approximations can be strongly preserving for properties if the removed behaviors do not influence the verification result. This is difficult to claim in practice when the under-approximation is generated by incomplete "coverage" of the program using a set of test cases or by an incomplete model-checking run. In this case, "coverage" is measured in terms of the program structure, which does not provide strong evidence to argue for preservation of many classes of properties. Typical model checking optimization techniques such as partial order and symmetry reductions (Clarke, Grumberg, and Peled 2000) are strongly preserving under-approximations.

Another relevant technique is *program slicing* (Corbett, Dwyer, and Hatcliff 2000). Slicing reduces the behaviors of a program by removing control points, variables, and data structures that are deemed irrelevant to checking a given property. Given a program and some statements of interest from the program, called the *slicing criteria*, a program slicer computes a reduced version of the program by removing program statements that do not affect the computation at the statements in the criterion. When checking a program against a property P, a slicing algorithm removes all the program statements that do not affect the satisfaction of P. Thus the property P holds on the reduced program if and only if it also holds on the original program (that is., slicing computes an exact abstraction). Program slicing is provided by analysis tools such as SPIN and Bandera.

4.2 Data Abstraction

Our presentation for data abstraction follows the abstract interpretation framework (Cousot and Cousot 1999). Abstract interpretation establishes a rigorous methodology for building data abstractions that are weakly preserving with respect to safety properties. In this framework, an abstraction maps the set of values of a program data type to an abstract domain and it maps the operations on that data type to functions over abstract values. Thus, the abstract program has a non-standard execution.

Typically, the behaviors of the abstracted program are an over-approximation of the behaviors of the original program; each executable behavior in the original program is "covered" by a behavior in the abstract program. Thus, the abstraction is weakly preserving. When the abstract behaviors are checked with a model checker against a safety property (or a property written in LTL) and found to be in conformance, we can be sure that the true executable behaviors of the unabstracted program satisfy the property.

4.2.1 Abstract Interpretation

We present an abstract interpretation in an informal manner, as a collection of three components:

1. A finite domain of abstract values;
2. An abstraction function that maps concrete program values to abstract values; and
3. A collection of primitive operations (one for each operation in the program).

Substituting concrete operations applied to selected program variables with corresponding operations of an abstract interpretation yields an abstract program.

Example

Consider a simple property that you may wish to verify: `assert(x==0);` for a program that contains integer variable `x`. Deciding whether this assertion is violated or not does not require complete knowledge about the value of `x`; you only need to know if `x` is zero or not. In this case, you can use the *Signs* abstraction which only keeps track of the sign of whether `x` is zero, negative, or positive.

We can define an abstract interpretation in the following way. The abstract domain is *{zero, pos, neg}*. The abstraction function maps integer value zero to *zero*, all positive integers to *pos*, and all negative integers to *neg*. We also need to define abstract versions of each of the basic operations on integers. For example, here are the abstract versions of + and ≤ operations on integers:

$+_{abs}$	*zero*	*pos*	*neg*
zero	*{zero}*	*{pos}*	*{neg}*
pos	*{pos}*	*{pos}*	*{zero,pos,neg}*
neg	*{neg}*	*{zero,pos,neg}*	*{neg}*

$\leq_{abs}$	*zero*	*pos*	*neg*
zero	*{true}*	*{true}*	*{false}*
pos	*{false}*	*{true,false}*	*{false}*
neg	*{true}*	*{true}*	*{true,false}*

Note that the return of multiple values in cases such as *neg* $+_{abs}$ *pos* ≡ *{zero, pos, neg}* models the lack of knowledge about specific the specific concrete values that these abstract tokens represent. Sometimes this is modeled with an explicit *Top* value that represents all the possible concrete values. This imprecision is interpreted in the model checker as a non-deterministic choice over the values in the set. Such cases are a source of "extra behaviors" that result from the abstract model over-approximating the set of real execution behaviors of the system.

In the following, we show excerpts of the Java representation for the *Signs* abstraction (as implemented by the Bandera tool (Corbett *et al.* 2000)).

```
public class Signs {
   public static final int ZERO = 0;
   public static final int POS = 1;
   public static final int NEG = 2;

   public synchronized static int abs_map(int n) {
      if (n == 0) return ZERO;
      if (n > 0) return POS;
      if (n < 0) return NEG;
   }

   public synchronized static int add(int a, int b) {
      if (a == ZERO && b == ZERO) return ZERO;
      if ((a == ZERO && b == POS) ||
         (a == POS && b == ZERO) ||
         ((a == POS && b == POS))
            return POS;
      if ((a == ZERO && b == NEG) ||
         (a == NEG && b == ZERO) ||
         (a == NEG && b == NEG))
            return NEG;
      return chooseInt(3);
   }
     ...
}
```

Here, abstract values are implemented as integer static fields, and the abstraction mapping and abstract operations are Java methods. Nondeterministic choice is specified as calls to `chooseInt(n)`, which is trapped by a model checker and returns non-deterministically values between 0 and `n`-1. The translation of a concrete program into an abstract program is straightforward: First select the program data that needs to be abstracted, and then replace concrete literals and operations with calls to classes that implement the abstract literals and operations. For example, here is a simple application that is abstracted using the *Signs* abstraction.

Concrete (un-abstracted) code

```
 class Application {
  public static void main(…){
   int x=0;
   int y=0;
   while(true) {
     x=x*y ;
     y++ ;
   }
   assert (x == 0);
  }
}
```

Abstracted code

```
class Application {
 public static void main(…) {
  int x=Signs.ZERO;
  int y=Signs.ZERO ;
   while(true)){
    x=Signs.mul(x,y);
    y=Signs.add(y,Signs.abs_map(1))
   }
   assert(x.eq(Signs.ZERO));
    }
  }
```

The abstracted program is just another Java program that can be analyzed with a model checker such as JPF. Note that the original unabstracted code has an unbounded number of states, since variable `y` is incremented in an infinite loop. However, the abstracted program is finite state and you can use JPF to prove that indeed there are no assertion violations for the analyzed code.

In order to guarantee that the behaviors of the abstracted program are indeed a superset of the behaviors of the unabstracted program, you must check that each of the abstract operations is an over-approximation of the corresponding unabstracted operation. For example, the following definition for $+_{abs}$ in the *Signs* abstraction: *neg* $+_{abs}$ *pos* ≡ *{zero, pos, neg}* is not an over-approximation of the un-abstracted "+" since it does not cover the cases like -2 + 2 = 0 when adding a negative and a positive number yields the result zero. You can use sophisticated techniques such as theorem proving or decision procedures to perform these checks automatically

4.2.2 Useful Abstractions

We discuss here several useful data abstractions for integer domains. Abstractions for other numeric domains can be defined similarly.

- A *Range* abstraction tracks concrete values between lower and upper bounds *l* and *u* and abstracts away the rest of the values. The abstract domain is *{below_l, l, …, u, above_u}*.
- The *Signs* abstraction is a special case of the Range abstraction where *l=u=0*.
- A *Set* abstraction can be used instead of a Range abstraction when only equality operations are performed, as when integers are used to simulate an enumerated type. For example, a *Set* abstraction that tracks only the concrete values 0 and 2 will have the abstract domain *{zero, two, other}*.
- A *Modulo* abstraction merges all integers that have the same remainder when divided by a given value.
- The *EvenOdd* abstraction with abstract domain *{even, odd}* is a *Modulo 2* abstraction.
- Finally, the *Point* abstraction collapses all integer values into an abstract value *unknown*. This abstraction is effectively throwing away all the information about a

variable, and it is useful when abstracting variables that have no significant effect on the property under analysis.

All these abstractions are supported by Bandera.

4.2.3 Abstracting Non-base Types

So far, we have discussed techniques for abstracting base types. Object-oriented programming languages offer some additional opportunities to apply abstraction. We discuss some of them below.

4.2.3.1 Abstractions for Data Structures

Abstraction of data structures such as Java classes can be achieved by component-wise abstraction of each field in a class. This approach is taken by Bandera (Dwyer *et al.* 2000). Array abstractions can be defined in a similar way: the user needs to define an integer abstraction for the array index and a data structure abstraction for the component type.

Manually created abstractions with this structure have been used for the verification of wireless protocols and are described in (Dams 1996).

4.2.3.2 Data Independence

A system is data independent if the values of the infinite program data it manipulates are not relevant to the behavior of the system—for example, a protocol that transmits messages may be proved correct regardless of the contents of the messages that are transmitted. It has been shown (Wolper 1985) that reasoning about pair-wise message ordering properties over a communication channel that accepts large domains of values can be achieved by using an abstract domain of size three. Note that this kind of reasoning is possible only when the values that are passed through the channel are not modified or tested by the program (i.e., the program is data independent). The three-valued abstract domain $\{d_1, d_2, other\}$ provides the ability to distinguish between two concrete elements, but it abstracts from their values. Such an abstraction can be thought of as a "symbolic" *Set* abstraction, where d_1 and d_2 are placeholder for *any* two different values of a particular concrete data domain, while all the other values are abstracted to *other*.

4.2.3.3 Container Abstractions

Abstractions for containers (such as lists, stacks, sets, or queues) may represent just the state of a container—e.g., full or empty—and abstract away from the actual container content. The list operations also need to be abstracted. For example, consider the operation `insert(l,x)` which adds element `x` to the end of list `l`. The corresponding abstract operation will have the following definition:

`Insert(`*full*`, x) ≡ Insert(`*empty*`, x) ≡` *full*

You can also use container abstractions in conjunction with the symbolic data abstractions presented above. Consider, for example, an abstraction that supports reasoning about ordered lists. This abstraction represents the behavior of lists that contain elements which were themselves abstracted using the abstract domain *{d_1, d_2, other}*. Conceptually, the abstract list records whether the list contains a specific d_1 value; the list also records the ordering of two inserted elements d_1 and d_2, but the abstraction does not record any information about the size of the list. The abstract list domain is the following: *{empty, some, [d_1], [d_2],[d_1, d_2], [d_2, d_1], all}*.

- *empty* represents empty lists.
- *some* represents lists with only other values.
- *[d_1]* represents lists that contain d_1 mixed with zero or more other values.
- *[d_2]* represents lists that contain d_2 mixed with zero or more other values.
- *[d_1, d_2]* represents lists that contain d_1 and d_2 in this order, plus zero or more other values.
- *[d_2, d_1]* represents lists that contain d_2 and d_1 in this order, plus zero or more other values.
- *all* represents lists that contain multiple d_1 and d_2 elements.

Using this abstraction, you can check ordering properties—for example, that if d_1 and d_2 are inserted in this order in a list, and then they are removed, they are removed in the same order. Similar abstractions have been used for checking ordering properties of protocols and memories (see Bensalem, Lackhneck, and Owre 1998; Graf 1999).

4.2.3.4 Heap Abstractions

The class abstractions that we discussed above are obtained by abstracting each field of base type. The number of instances of that particular class still needs to be *bounded*; this results in an under-approximation that is still useful for finding bugs. *Heap abstractions*, on the other hand, abstract the portion of the heap that stores instances of one particular class; therefore, heap abstractions can handle an *unbounded* number of dynamically allocated objects. These abstractions represent heap cells by *shape nodes* and sets of indistinguishable runtime locations by a single shape node, called a *summary node*. For example, a heap abstraction will keep concrete information about memory cells that are directly pointed to by local variables and merge all the memory cells that are not pointed to by a program variable into a summary heap node. Heap abstractions are most often used in static analysis tools (see TVLA website), but they are starting to be used in software model-checking approaches.

4.3 Predicate Abstraction

Predicate abstraction (Graf and Saïdi 1997), is a special case of an over-approximating data abstraction which maps a potentially infinite-state program into a finite-state program, via a finite set of predicates over the program's variables. Predicate abstraction allows you to build an abstract program in a fully automatic way, and it forms the basis of numerous automated abstract model-checking tools (Ball *et al.* 2001; Henzinger *et al.* 2002; Chaki *et al.* 2004). The basic idea is to replace a concrete program variable by a Boolean variable that evaluates to a given Boolean formula (a predicate) over the original variable. This concept can be easily extended to handle multiple predicates and, more interestingly, predicates over multiple variables. For example, assume we have a program with two integer variables, `x` and `y`, which can grow infinitely. This program will have an infinite state space and model checking cannot be complete in general. However, if the only relationship of interest between the two variables is their equality, we can use a predicate representing this relationship, `B≡ x==y`, to construct an abstract program as follows. (We used similar abstraction when analyzing DEOS in section 4.5.)

1. wherever the condition `x==y` appears in the program, we replace it with the condition `B==true`, and
2. whenever there is an operation involving `x` or `y`, we replace it with an operation changing the value of `B` appropriately.

When generating the abstract program, over-approximation can occur when not enough information is available to calculate a deterministic next action in the abstract program. For example, consider the following concrete program instruction:

```
x := x + 1;
```

Assume that in the abstract program, `B` is false (`x!=y`). In this case, `B` can become non-deterministically true or false as a result of executing the instruction, because there is not enough information encoded in the initial value of `B` (specifically, we do not know whether `y==x-1`). On the other hand, if `B` is true (`x==y`) before executing `x:=x+1`, then we know for sure that `B` becomes false after execution. Therefore, the abstract version of the instruction with respect to `B` is the following:

```
if (B) // x==y
   B := false;
else // x!=y
   B := chooseBool();
```

The abstract version of a program can be computed in a semi-automatic way, using off-the-shelf decision procedures (or theorem provers). Investigating approaches to automatically select and refine abstractions is an active research area. The problem of constraining the over-abstraction is analogous to the problems encountered when attempting to make static analysis more precise.

Counter-Example Guided Abstraction Refinement (the CEGAR loop) is a popular approach to automatic abstraction and refinement. Recall that due to over-approximation introduced by the abstraction, reported counterexamples may be spurious. The CEGAR method analyzes abstract counterexamples to see if they correspond to real errors. If they are spurious, the abstraction is refined automatically:

New abstraction predicates are added and the abstraction is computed again, such that the previously reported spurious counterexample is eliminated. Counterexample analysis uses weakest precondition calculations and decision procedures or theorem proving. The CEGAR loop leads to fully Automatic Predicate Abstraction.

4.4 Model-Driven Verification

So far we have looked at data abstraction techniques that compute over-approximations of programs. We discuss briefly here a complementary technique, model-driven verification, which analyzes an under-approximation of a system. The presentation in this section follows (Holzmann and Joshi 2004).

Model-driven verification, as implemented in the SPIN tool, advocates the use of abstraction mappings during concrete model checking to efficiently analyze an under-approximation of the feasible behaviors of a system. All reported counterexamples correspond to real errors. The model checker traverses the concrete transitions of the analyzed system and for each explored concrete state it stores an abstract version of that state. The abstract state, computed by an abstraction mapping provided by the user, is used to determine whether the model checker's search should continue or backtrack (if the abstract state has been visited before). This allows for detection of subtle errors which cannot be discovered with classic techniques. Model-driven verification has been applied to the verification of a NASA Mars Exploration Rover (MER) Arbiter (Holzmann and Joshi 2004) (see section 9.5). The technique has been subsequently extended with automatic predicate abstraction and refinement (Pasareanu, Pelanek. and Visser 2005), and applied to the verification of a component extracted from an embedded spacecraft control application that has deadlocked in space—the NASA Deep Space 1 Remote Agent Experiment.

SPIN v4 provides the capability of including a software application (written in C) without substantial change into a verification test harness written in Promela, and then verifying it directly, while preserving the ability to perform data abstraction. The test harness is used to drive the application through all its relevant states. Several Promela primitives can be used to connect a verification model to implementation-level C code:

- `c_decl` introduces the types and names of externally declared C data objects that are referenced in the model.
- `c_track` defines the data objects that appear in the embedded C code that should be considered to hold state information during model checking.
- `c_code` encloses an arbitrary fragment of C code.
- `c_expr` evaluates a C expression to a Boolean value.

Consider the following small example (taken from Holzmann and Joshi 2004):

```
c_decl {
   extern float x;
   extern void fiddle(void);
};
```

```
c_track "&x" "sizeof(float)";
init {
   do
   :: c_expr { x < 10.0 } -> c_code { fiddle(); }
   :: else -> break
   od
}
```

Here `c_decl` introduces definitions of externally declared variable `x` and function `fiddle()`; `c_track` indicates to the model checker that variable `x` holds state information that must be tracked (it provides the address of the variable and its size in bytes). The `init` process repeatedly calls the external function `fiddle()`, while it sees that the value of `x` is less than 10.0. Note that the `c_track` primitive supports two separate goals: state tracking and state matching.

There are cases where the value of an external data object should be tracked, to allow the model checker to restore the value of these data objects when backtracking, but where the data object does not actually hold relevant state information. In other cases, the data object does hold state information, but contains too much detail. In both of these situations you can define abstractions on the data that are used in state-matching operations, while retaining all details that are necessary to restore state in the application in tracking operations. An additional, and optional, argument to the `c_track` primitive specifies whether the data object should be matched in the state space. There are two versions:

```
c_track "&x" "sizeof(float)" "Matched";
```

and

```
c_track "&x" "sizeof(float)" "UnMatched";
```

with the first of these two being the default if no third argument is provided. The value of unmatched data objects is saved on the search stack, but not in the state descriptor. The simplest use of this extra argument is for tracking data without storing it in the model checker's searched state space. When used in this way, the use of unmatched `c_track` primitives equates to data hiding. Another use is for unmatched `c_track` statements to hide the values of selected data objects and then to add abstraction functions (implemented in C) to compute abstract representations of the data that will now be matched by he model checker.

As a simple example, consider two integer variables `x` and `y` that appear in an application program. Suppose that the absolute value of these two variables is irrelevant to the verification attempt, but the fact that the sum of these two variables is odd or even *is* relevant. We can setup a data abstraction for this application as follows:

```
/* data hiding */
c_track "&x" "sizeof(int)" "UnMatched";
c_track "&y" "sizeof(int)" "UnMatched";
/* abstraction: */
c_track "&sumxy" "sizeof(unsigned char)" "Matched";
```

and we add the abstraction function:

```
c_code {
void abstraction(void) { sumxy = (x+y)%2; }
}
```

which must be called after each state transition that is made through calls on the application-level code.

4.5 Tool Support and Applications

As we have mentioned, Bandera supports data abstraction and SPIN supports model-driven verification. SLAM (from Microsoft) performs predicate abstraction with counterexample-based automatic refinement. The SLAM analysis engine has been incorporated into a new tool Static Driver verifier (SDV), which is being used within Microsoft for checking Windows device drivers. It is currently in beta form as part of the Windows Driver Development Kit. BLAST (developed at UC Berkeley) and MAGIC (developed at Carnegie Mellon University) (MAGIC website) are freely available software analysis tools that also use predicate abstraction for the analysis of C code. See sections 7.4 and 7.9 for information about BLAST and SLAM.

Abstraction has been proved essential for the verification of diverse software, ranging from Windows device drivers to flight software. We describe here our experiment with using abstraction for the automated verification of the Honeywell DEOS realtime scheduling kernel (see section 9.3). The experiment involved translating a core slice of the DEOS scheduling kernel from C++ into Promela, constructing a test driver, and carefully introducing data abstraction to support exhaustive verification. Verification of several properties led to the rediscovery of a known error in the implementation.

In order to allow exhaustive verification of the DEOS model, it was necessary to apply abstraction to the program. Our goal was to introduce abstraction carefully and selectively to increase our understanding of how abstraction is applied during verification. In this section, we relate the steps we took to first introduce a simple, ad-hoc abstraction into the system, and then to refine the abstraction using predicate abstraction.

In the abstraction process, we were guided by several experiments showing traces through the system that were 2,000,000 steps long. Based upon our limited intuition of how DEOS works, this seemed too long because the system’s behavior is cyclic in nature. These extremely long traces indicated that some data was being carried over multiple cycles of computation. We were able to identify this data simply by running a simulation and observing the SPIN data values panel. In particular, we identified a data member, `itsPeriodId`, in class `StartOfPeriodEvent`, which was operating as a counter, incrementing in a potentially infinite loop. An additional variable, `itsLastExecution`, was also incrementing, because it was periodically assigned the value of the `itsPeriodId` counter.

The section of the DEOS kernel involving `itsPeriodId` and `itsLastExecution` is shown below.

```
void StartOfPeriodEvent::pulseEvent( DWORD systemTickCount ) {
   countDown = countDown - 1;
   if ( countDown == 0 ) {
      itsPeriodId = itsPeriodId + 1;
      ...
      }
void Thread::startChargingCPUTime() {
   // Cache current period for multiple uses
   periodIdentification cp = itsPeriodicEvent->currentPeriod();
      ...
   // Has this thread run in this period?
   if ( cp == itsLastExecution ) {
      // Not a new period. Use whatever budget is remaining.
      ...
   }
   else {
      // New period, get fresh budgets.
      ...
      // Record that we have run in this period.
      itsLastExecution = cp;
      ...
   }
   ...
}
```

`itsPeriodId` and `itsLastExecution` are used to determine whether or not a thread has executed in the current period of time. The two variables are either equal or different by one. After consulting with the DEOS developers, we were assured that the version of DEOS we were considering does in fact ensure that each thread is scheduled every period, and hence we changed `itsPeriodId` to be incremented modulo 2. This change led to an under-approximating abstraction that allowed exhaustive analysis of the entire state space of the buggy version of DEOS as well as the corrected version. We also used predicate abstraction, using a predicate that encodes a relationship between `itsPeriodId` and `itsLastExecution` (the predicate records whether they are equal or not). As explained in (Penix *et al.* 2005), this abstraction is in fact exact (i.e., there is no loss of precision due to abstraction).

5 Search and Partial Coverage

A model checker can be used to verify the correctness of a finite-state system with respect to a desired property by searching a labeled state-transition graph that models the system to see whether it satisfies the property specified in some notation (e.g., as a Linear Temporal Logic (LTL) formula).

After running a model checker on the target application model there can be two main outcomes. Either the tool finds one or more violations of the specified property or it does not. If it finds such a violation, the tool typically returns a counterexample—a trace of program execution—which shows how it occurred. The user can then use the results to locate and fix the problem.

On the other hand, the model checker might not find such violations. Instead, it might:

- Finish and report that its work is complete and no violations were detected;
- Die before completing because it ran out of system resources such as memory; or
- Get lost in the vast state space and seem to be running indefinitely (the execution takes too long with no sign of the light at the end of the tunnel).

We have seen that the primary challenge in program model checking is the state space explosion problem. Exploring all the behaviors of a system is, to say the least, difficult when the number of behaviors in the possible inputs, contents of data structures, or number of threads in the program is exponential. Even with the right models and property specifications, it is not possible in practice cover the entire state space of most realistic systems and therefore result (1) is unlikely. Quite often the best we can do is to try to maximize our *partial coverage* of the state space as far as possible.

The good news is that models and property specifications are not the only input parameters for a model checker. Rather than blindly search through the state space using some fixed search strategy, as is common for traditional model checking, program model checkers can be configured to maximize their effectiveness in checking the desired properties during a particular run. For example, different search strategies and heuristics can be selected to guide the model checker as it navigates through the maze of the state space to focus on the parts that are most likely to contain bugs first—and there lies the real art of model checking.

When a model checker does not find any violations of the specified properties, it typically does not return any information beyond an indication that the work was complete or that some runtime exception occurred. It would be nice to get some *coverage metrics* to indicate which parts of the state space were covered during the analysis. This information can also be used as heuristics in future runs of the model checker to achieve better coverage of the state space.

In section 5.1 we describe some of the searching strategies that are commonly used in software model checking. Section 5.2 describes the relationship between model checking and testing. In section 5.3 we present several ways in which program model checking can be improved by taking advantage of the close relationship to testing, and discuss structural coverage measures from testing that can measure partial coverage by model checking. In section 5.4 we discuss how the capability to calculate structural coverage (e.g., branch coverage) can be used by the model checker for guided search.

5.1 Selecting a Search Strategy

Model checkers such as JPF and SPIN support a number of search strategies used to explore the state space of the program.

Two of these strategies are the most well-known—Depth-First Search and Breadth-First Search, which are discussed below.

5.1.1 Depth-First Search

With *Depth-First Search* (DFS), the model checker begins at the start state and explores each of its outgoing transitions to other states as far as possible along each branch before backtracking and selecting the next transition. DFS goes deeper and deeper until a goal state is reached (e.g., an error state such as a deadlock) or it hits a state with no outgoing or unexplored transition (e.g., an end state). The search backtracks, returning to the most recent state with unexplored transitions. A stack is usually used to hold the frontier (the set of states that are still in the process of being explored), and therefore the last state added to it is always selected first during the search.

5.1.2 Breadth-First Search

With *Breadth-First Search* (BFS), the model checker again begins at the start state and explores all its outgoing transitions to the neighboring states. Then for each of those nearest states, it explores all of *its* outgoing transitions to its neighbor states, and so on, until it hits a goal state or a state with no outgoing or unexplored transition. A queue is usually used to hold the frontier, and therefore the first state added to it is always selected first during the search.

5.1.3 DFS and BFS Tradeoffs

In general, DFS will have lower memory (space) complexity than BFS since only the information about the states and transitions on the current path needs to be stored. This contrasts with BFS, where all the states and transitions that have so far been explored must be stored.

One disadvantage of DFS is that it can sometimes be bad for infinite or very large state spaces. A search that makes the wrong first transition choice can search for a long and indefinite period of time with no success. This is particularly true if a state has a successor that is also one of its ancestors and the model checker is not configured to check for visited states. The model checker easily can run out of memory before it gets to an answer.

With BFS, the model checker will not get trapped exploring a blind alley. It is guaranteed to find the error state with the shortest path from the root, since all parts of the state space are examined to level *n* before any states on level *n+1* are examined. Furthermore, if there are multiple error states, the error state that requires a minimum number of transitions from the start state will be found first. This contrasts with DFS, which may find a long path to the error state in one part of the state space, when a shorter path exists in some other, unexplored part of the state space. If the number of neighboring states (also called *branching factor*) is too large, the DFS model checker may run out of memory before it can reach larger depths in the state space where the interesting states may be hiding. Of course, if the model checker is configured to store states regardless (for example to avoid checking the previously visited states again) then memory consumption would not be different in the two cases.

5.1.4 Bounded Searches

So far we have looked at two main search algorithms that can in principle be used to systematically search the whole state space. But the state space may be just too big for such strategies to find what we are looking for before the model checker runs out of space. To alleviate, this problem we can use bounded versions of the DFS and BFS search strategies.

Bounded Depth-First Search (BDFS) works exactly like Depth-First Search, but avoids some of its drawbacks by imposing a maximum limit on the depth of the search. Even if the search is capable of exploring transitions to states beyond that depth, it will not do so and thereby it will not follow infinitely deep paths or get stuck in cycles. Therefore, it can find a solution only if it is within the depth limit. For essentially infinite-state systems (e.g., in our DEOS case study) limiting the depth is the only practical way to use DFS, but finding the proper depth can be difficult and large depths may result in extremely long counterexamples.

In practice, you can run the model checker with BDFS strategy multiple times, each time with ever-increasing depth limits (0, 1, 2, …). This is called *Iterative Deepening Depth-First Search*. This way we still have the advantage over BFS that space complexity is *O(n)*, and we have the advantage over DFS that we are guaranteed to find the solution with the shortest path from the root. We can also show that the time complexity of iterative deepening is not much worse than for breadth- or depth-first search.

Similarly, a *Bounded Breadth-First Search* (BBFS) works exactly like Breadth-First Search, but it imposes a maximum limit on the number of neighboring states that are searched next.

5.1.5 Heuristic Search

Heuristic search exploits the information known about the system under analysis in order to accelerate the search process. The basic idea of heuristic search is that rather than trying all the possible search paths, the model checker tries to focus on paths that seem to be getting it nearer to the error states. This is a strong motivation for the use of heuristic search in bug finding; model checkers which rely on traditional DFS algorithms tend to return much longer counterexamples than those which support heuristic search. Of course, the model checker which uses heuristic search generally can't be sure that it is really close to an error state, but it might be able to have a good guess. Heuristics are used to help the model checker make that guess.

Different search algorithms can be combined with heuristics. To use heuristic search, we need an evaluation function that scores a state in the state space according to how close to the error state it seems to be—the higher the score, the more desirable the state is to be explored next.

In general, the heuristic evaluation function may depend on many factors:

- The description of the next state
- The description of the goal state
- The information gathered by the search up to that point
- Any extra knowledge about the problem domain

Model checkers such as Java PathFinder allow users to extend the existing set of heuristics that come with the tool with user-defined heuristics. JPF provides full access to its internal state through an API that can be used to define powerful and complex heuristics. In practice, users can combine different heuristics which are used during the search. As a result many search algorithms exist and their full description is beyond the scope of this guidebook. We will only cover some of the most common algorithms that we have found useful in our case studies.

5.1.5.1 Best-First Search

Best-first search is a variant of BFS which improves it by exploring the most promising neighboring state chosen according to the result of the evaluation function for that state. Efficient selection of the current best candidate for exploration is typically implemented using a priority queue.

There are different ways to compute the evaluation function, and therefore there are different variants and refinements of best-first search. Some of those described here are Greedy Best-First Search, A* Search, and Beam search.

5.1.5.1.1 Greedy Best-First Search

Greedy best-first search uses a heuristic that attempts to predict how close the end of a path is to the goal state, so that paths which are judged to be closer to the state are explored first.

5.1.5.1.2 A* Search

*A** (pronounced "A star") is a best-first search that attempts to minimize the total length of the path from the initial state to the goal state. It combines advantages of breadth-first search, where the shortest path is found first, with those of best-first search, where the state that we guess is closest to the goal state is explored next.

The heuristic function score, *f(s)*, for a state s is calculated as the sum of the length of the path so far, *g(s)*, plus the estimated length to the goal state, *h(s)*:

$$f(s) = g(s) + h(s)$$

This search strategy is particularly attractive for verification purposes where we want to get the shortest error trace to a bug. We may often use other simple-minded search strategies to find a bug, but then use A* to get the shortest path to it.

5.1.5.1.3 Beam Search

Beam search is an optimization of the best-first search that uses the heuristic function to discard all but the *m* best candidate states at each depth, where *m* is the bound on the "width of the beam." If more than *m* candidates exist, the worst candidates are discarded. This reduces the space requirements of best first search. Assigning a bound to the width of the search can also be applied to best-first, greedy best-first, and A* searches.

When such a bound is used, termination of the search without discovering an error does not imply correctness of the system. However given that the advantage of heuristic search is its ability to quickly discover fairly short counterexamples, in practice use of such a bound is an effective bug-finding tactic.

We may have to run the model checker multiple times each time with a different and increasing value for *m*. This is called iterative widening.

5.1.5.2 Random Search

Random search non-deterministically selects a state on the frontier to explore, and can find results even when exhaustive search is not feasible. Its success in artificial intelligence makes it a good candidate search strategy for model checking, but its application obviously limits the rigor of verification, because faults can hide in portions of the state space not explored in random trials. Random search should be used when the objective is more bug-finding than verification. For high-assurance systems, its usage is recommended earlier in the development and debugging when the number of logical faults is higher than in the later stages and when their removal is the most cost effective.

We experimented with using a purely random search to find a known bug in DEOS using JPF; however, the counterexample was considerably longer and took more time and memory to produce than with using the coverage heuristics (described later). See also section 7.7 about the LURCH system, which employs random search and has performed well in certain cases.

5.2 Model Checking vs. Testing

Model checking is often claimed to have an advantage over testing since all possible behaviors of a system are analyzed, the idea being that model checking might catch subtle errors that testing could miss (Figure 4).

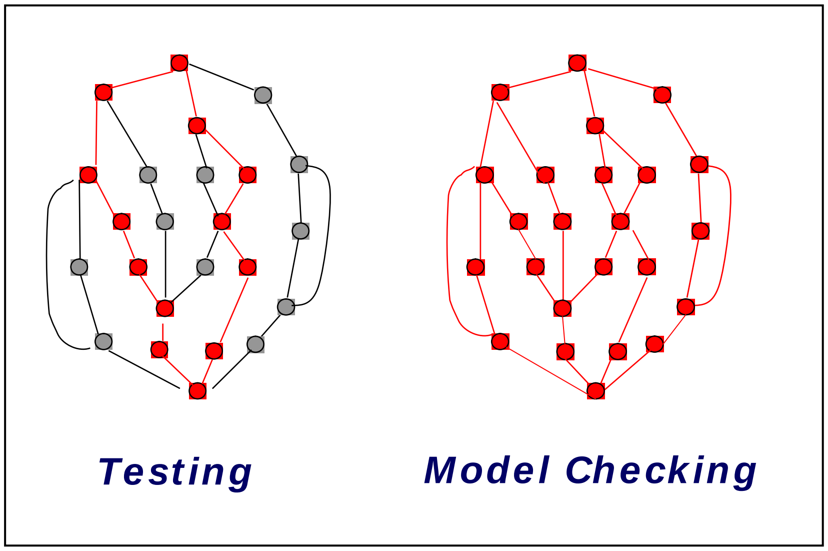


Figure 4 Model Checking vs. Testing

The first work linking model checking and testing was to use model checking for test-case generation (Callahan, Schneider, and Easterbrook, 1996; Gargantini and Heitmeyer 1999).

One notable difference between testing and model checking is that model checking is more suited to analysis of concurrent and reactive programs because it has full control over the scheduling of processes, threads, and events, which is not typically the case in testing. Also, because a model checker stores the program states that it has visited (to stop it from getting lost by repeating itself), it can be more efficient in covering the behaviors of a program (Park *et al.* 2000).

While this is true in theory, real systems tend to have large state spaces, and it is impossible to store all the states of the system. That is the bad news. The good news is that, in case studies involving real systems, errors can usually be revealed by considering just a few subtle interactions, rather than a multitude of complex ones. Looking at only part of the state space (or behaviors) can therefore be effective for finding errors when using a model checker to analyze a real program.

5.3 Coverage for Model Checking

When model checking is incomplete and no errors are reported, we would like to gain information about what aspects of the program's behavior have been checked. To do this, we can use some standard (and some not-so-standard) test coverage criteria.

Test coverage provides a measure of how well the testing framework actually checks the application software and how can we know when to stop testing. The situation with a model checker is similar. We want to know what parts of the state space were covered. There are different coverage levels that we can measure (Cornett 1996):

- Function Coverage
- Statement Coverage
- Decision Coverage
- Condition Coverage
- Multiple Condition Coverage
- Condition/Decision Coverage
- Modified Condition/Decision Coverage
- Path Coverage
- Branch Coverage

During testing, structural code coverage measures such as decision (or branch) coverage are used to obtain confidence that a program has been adequately tested. The reasoning is that if a program structure is not covered, then there is no basis to argue that that part of the program is error-free. For example, the Federal Aviation Administration requires software testing to achieve 100% modified condition/decision coverage (MC/DC) in order to certify level A criticality software for flight in RTCA document DO-178B (RTCA website). MC/DC coverage requires that all Boolean conditions within a decision independently affect the outcome of the decision. It was designed to achieve branch coverage of object code (compiled source) and provide confidence that logic errors do not occur in a program.

Similar coverage measures can be used for model checking. On the face of it you might wonder why coverage during model checking is of any value, since model checkers typically cover all of the state space of the system under analysis, hence by definition covering all the structure of the code. However, calculating coverage serves the same purpose as during testing: it shows the adequacy of the (partial) model checking run. The output of a model checker now indicates either that an error was found and provides a path that shows how to get to it, or, if no error is found, returns a coverage measure that testing engineers can interpret as is done now.

The real question is figuring out why this might be useful. A model checker that can calculate traditional structural coverage could help answer at least one interesting question that many people, including the FAA, have been struggling with for some time: How good is the MC/DC coverage measure at finding safety-critical errors? (Dupuy and Leveson 2000). In general, it is not known whether a partial model-checking result that includes 100% MC/DC coverage provides much of a guarantee of error-free operation. Note that achieving a certain structural coverage is not known to be useful in finding certain types of behavioral errors, such as timing/scheduling errors—exactly the ones model checking is good at finding.

Model checkers are particularly suited to finding errors in concurrent programs, but many traditional coverage criteria were developed for sequential programs. Therefore, it is not clear that tracking coverage of these metrics during model checking would be useful. It may be more appropriate to investigate coverage measures more suitable to concurrent systems, such as all-DU-paths for concurrent programs (Yang, Souter, and Pollock 1998).

A more interesting approach may be to develop suitable behavioral coverage measures. For example, "relevant path coverage" might be used to indicate coverage of the paths relevant to proving a property correct. Using behavior-based coverage metrics, it should be clear that program abstraction techniques, such as slicing and conservative abstraction, still provide full coverage even though some paths are not completely checked.

A model checker, during normal execution, can calculate coverage metrics that can be used to evaluate how the paths executed by the model checker rate against established testing criteria. For example, the SPIN model checker calculates statement coverage and reports unreachable code.

5.4 Directed Model Checking

Directed model checking addresses the state explosion problem by using guided or heuristic search techniques during state space exploration (Edelkamp, Lluch-Lafuente, and Lee 2004; Edelkamp 2006). It borrows heavily from well-known AI techniques to enable the model checker to better home in on error states while searching just a portion of the state space, and thereby avoid the problem of state space explosion. Here we will describe different types of heuristics that may be used during directed model checking.

5.4.1 Property-Specific Heuristics

Property-specific heuristics are based on specific properties such as:

- Deadlock: Maximize number of blocked threads
- Assertions and exceptions: Minimize distance to assertions and throws of explicit exceptions
- Language-specific size constraints such as overflow of a particular buffer
- Resource consumption: e.g., attempts to exceed the limit on open file handles

A lot of the work on model checking using heuristics largely concentrates on property-based heuristics. Selecting such heuristics in practice is not easy, particularly when the model checker is applied to a large concurrent program with many types of errors (e.g., assertions, potential for deadlocks, uncaught exceptions). Also, it is not always possible to know during model checking how close you are to a property violation.

5.4.2 Structural Heuristics

Structural coverage measures can be used during directed model checking to improve the state space coverage (Groce and Visser 2002). Rather than looking for a particular error, it may be best to try and explore the structure of the program systematically, looking for any kind of error. As noted before, this is the motivation behind coverage metrics in testing. For example, you can guide the model checker to pick parts of the state space to analyze based on structural coverage of the code that would generate those state spaces. A simple example would be to consider only statement coverage: If the model checker can next analyze a program statement that has never executed, versus one that has, then it picks the new one. This simple example is a greedy algorithm which may not work. We will describe some of the structural heuristics that have been found to be useful for model checking.

5.4.2.1 Code Coverage Heuristics

In many industries, 100% branch coverage is considered a minimum requirement for test adequacy (Bezier 1990). Branch coverage requires that at every branching point in the program all possible branches be taken at least once. Calculating branch coverage during model checking can be done by keeping track of whether at each branch point all options were taken. JPF goes further than that. It also keeps track of how many times each branch was taken and considers coverage separately for each thread created during the execution of the program. JPF can produce detailed coverage information when it exhausts memory without finding a counterexample or searching the entire state space. You can use this data as heuristic to prioritize the exploration of the state space.

One approach to using branch coverage metrics would be to simply use the percentage of branches covered (on a per-thread or global basis) as the heuristic value. This is referred to as the %-coverage heuristic. However this heuristic is likely to fall into local minima, exploring paths that cover a large number of branches but do not in the future increase coverage. Instead, a slightly more complex heuristic proves to be more useful:

- States covering a previously untaken branch receive the best heuristic value.
- States that do not involve taking a branch receive the next best heuristic value.
- States covering a branch already taken can be ranked according to how many times that branch has been taken, with worse scores assigned to more frequently taken branches.

You can create many variations of such branch-counting heuristics. Counts can be taken globally (over the entire state space explored) or only for the path by which a particular state is reached. The counts over a path can be summed to reduce the search’s sensitivity to individual branch choices.

Such heuristics can be more sensitive to data values than coverage measures traditionally used in testing, since they go beyond the simple 0-1 sensitivity of all-or-nothing coverage.

Note also that the branch-counting heuristics can be used in dynamic test case generation (Korel 1990) by using the heuristic function to optimize the selection of test cases—for example, by only picking cases in which the coverage increases.

The practical effect of this class of heuristic is to increase exploration of portions of the state space in which nondeterministic choices or thread interleavings have resulted in the possibility of previously unexplored or less explored branches being taken. In theory, all of the variations of the branch-counting heuristic mentioned above can produce significantly different results on real programs, but in experimental results using JPF, only global vs. path had any observable impact. Storage of the visited states could also have a great impact on the success of such heuristic searches. While model checking DEOS we discovered that without state storage these heuristics failed to find a counterexample for the known bug before exhausting memory—the queue of states to explore became too large and exhausted the memory.

5.4.2.2 Thread Interleaving Heuristics

A different kind of structural coverage is based on maximizing thread interleavings. Traditional testing often misses subtle race conditions or deadlocks because generally the scheduler cannot be controlled directly. One way to expose concurrency errors is to reward "demonic" scheduling by assigning better heuristic values to states reached by paths involving more switching of threads. In this case, the structure we attempt to explore is the dependency of the threads on precise ordering. If a non-locked variable is accessed in a thread, for instance, and another thread can also access that variable (leading to a race condition that can result in a deadlock or assertion violation), that path will be preferred to one in which the accessing thread continues onwards, perhaps escaping the effects of the race condition by reading the just-altered value.

One heuristic that has been shown to be effective is calculated by keeping a (possibly size-limited) history of the threads scheduled on each path:

- At each step of execution, append the thread just executed to a thread history.
- Pass through this history, making the heuristic value that will be returned worse each time the thread just executed appears in the history by a value proportional to:
  - How far back in the history that execution is, and
  - The current number of live threads.

Application of JPF with this interleaving heuristic to the famous dining philosophers' example shows that this interleaving heuristic can scale to quite large number of threads. While DFS fails to uncover counterexamples even for small problem sizes, the interleaving heuristic can produce short counterexamples for up to 64 threads.

The key difference between a property-specific heuristic and a structural heuristic can be seen in the dining philosophers' example where we search for the well-known deadlock scenario. When increasing the number of philosophers high enough (e.g., 16) it becomes impossible for an explicit-state model checker to try all the possible combinations of actions to get to the deadlock, and heuristics (or luck) are required. A property-specific heuristic applicable here is to try and maximize the number of blocked threads, since if all threads are blocked we have a deadlock (most-blocked heuristic). Whereas a structural heuristic may be to observe that we are dealing here with a highly concurrent program—hence it may be argued that any error in it may well be related to an unexpected interleaving —we use the heuristic to favor increased interleaving during the search (interleaving heuristic). Although experimental results are by no means conclusive, it is still worth noting that for the dining philosophers' example the structural heuristic performs much better than the property-specific heuristic. Another interesting observation is that turning the state storage off with the interleaving heuristic mentioned above for both the dining philosophers' example and the Remote Agent example described in section 9.1 does not seem to change the results significantly—we found only minor variations in the length of counterexamples and number of states searched. In contrast, turning the state storage off for the most-blocked heuristic seems to cause previously successful searches to become unsuccessful.

5.4.2.3 Thread Preference Heuristic

This is similar to the thread interleaving heuristic mentioned above except that it focuses on a few threads that may be suspected to harbor an error. To do this, it relies on the knowledge of what parts of the system are "interesting." For example, when JPF was applied to the Remote Agent example, the race detection algorithm was used in a first run of the model checker with the BFS search strategy to identify a number of potential race conditions. Allowing the race detection to run for 3 minutes revealed that the Executive and Planner threads had unprotected write accesses to a field. The threads involved in the potential race conditions were then used to guide a thread-preference search with a similarly small queue, and a counterexample was quickly detected. This approach scaled to larger versions of the Remote Agent than other heuristics could handle. This is a different flavor of structural heuristic than those presented previously. It relies on specific knowledge of the program's behavior that can be observed by the model checker (which branches are taken, which threads are enabled, etc.) during the execution of the program. In certain cases, such as the one described above, this knowledge can be automatically extracted by the model checker itself. Such additional knowledge can, as expected, aid a guided search considerably.

5.4.2.4 Choose-free Heuristic

With the choose-free heuristic, the model checker first searches the part of state space that does not contain any non-deterministic choices. This is particularly useful when abstractions based on over-approximations of the system behavior are used for reducing the size of the state space to allow more efficient model checking. A variation of this heuristic gives the best heuristic values to the states with the fewest nondeterministic choice statements enabled, allowing the choose-free state space to be searched first but continuing to the rest of the state space otherwise (this also allows choose-free to be combined with other heuristics).

5.4.3 User-Guided Searches

Traditionally heuristics are often problem-specific. Structural heuristics and property-specific heuristics of general utility are provided as built-in features of model checkers such as JPF, but it is often essential to allow users to also craft their own heuristics. Model checkers like JPF allow users to define their own heuristics which can take advantage of the user’s special knowledge of the program being analyzed.

6 Program Design Guidelines for Model Checking

Most significant software development projects use some kind of design and coding guidelines. There exist many such guidelines, some including tens or even hundreds of rules covering almost all aspects of development—from the use of white spaces, naming conventions, and comments to recommendations on what specific language features to use or avoid (MISRA 2004; Meyers 2005; Sutter 1999; Sutter 2004). The choice of language and also the type of application that is developed obviously play an important part in the set of rules that are applicable in practice. Some rules may contradict or be inconsistent with other rules within the same guideline document. Therefore developers are often forced to follow some of the rules and ignore others. With no automated checking of compliance, many violations of these guidelines often go unnoticed.

There does not seem to be a consensus on what makes a good guideline in general. Even less obvious is the effect of some of these guidelines on software verifiability. (Holzmann 2006) proposes 10 rules that could have measurable effect on software reliability and verifiability. He argues that although the proposed small set of rules cannot be all-encompassing, setting an upper bound can increase the effectiveness of the guideline. In this section we describe a number of recommended design and coding guidelines which can be used effectively to tame the state explosion problem and also to reduce the amount of effort that is required to use a model checker to verify properties of interest.

Section 6.1 starts by listing various C/C++ programming language features and implementation idioms which expand the state space of a program. In that sense, these features (like multiple inheritance) are not necessarily indicative of bad design, but their use should be limited to enable model checking, and also to improve testability. It is also important to design applications to be state aware (e.g., avoid redundancy or unnecessary concurrency). Section 6.2 lists such state-reducing measures. These measures are by no means model-checking specific; they represent general "best design-practices" which have the side effect of making systems more suitable for model checking.

6.1 C/C++ Modeling and Translation Constraints

This section lists constructs and idioms in C and C++ which should be avoided because the corresponding models in other languages (e.g., Java, Promela) can have negative impacts on the memory consumption in general, and state space in particular. Some constructs are clearly considered to be outside of the modeled language (i.e., would be too expensive to model in a more abstract language like Java). The main focus is on state space efficiency; runtime efficiency, if addressed, is a secondary concern.

6.1.1 Pointer Arithmetic

C and C++ provide the ability to modify a pointer's target address with arithmetic operations. This is used, for example, to index arrays.

```
MyObject* p = …
…
p = p + 42;
```

For model checkers such as Java whose modeling notations do not support these operations directly, these operations can be problematic to model. For example a pointer variable can be mapped directly to a Java reference unless any pointer operations are performed on it anywhere in the code (e.g., pointer addition or taking its address). They require creation of explicit "C-pointer" objects with additional state (base, offset). We also need to create a model of the memory and the platform in order to be able to effectively model such arithmetic operations. While this can be supported in practice, it can significantly increase the complexity of the model and hence the size of the state space.

6.1.2 Primitive Type Casting

C/C++ allows you to cast between totally unrelated types. This can be problematic for model checking. Avoid type casting between unrelated types and in particular primitive types. For example,

```
long l;
…
char* c = (char*)&l;
…
c[i] = x;
```

Casting between unrelated types requires overhead for keeping values consistent. For primitive types, this also requires creating objects instead of primitives and explicit modeling of the memory and the platform. In languages like Java you cannot even use the immutable standard box types like `java.lang.Integer`. Furthermore, operations on the primitives have to be turned from direct bytecode instructions into method calls, creating more overhead.

6.1.3 Unions

Unions in C and C++ are object overlays—aggregate quantities like structs, except that each element of the union has offset 0, and the total size of the union is only as large as is required to hold its largest member (Kernighan and Ritchie 1998).

```
union U {
  unsigned long
l;
  char c[4];
}

U u;
u.l = X;
…
u.c[0] = Y;
```

Unions are not supported by languages like Java. The translation has to generate different objects for the union itself and all its fields, plus the code to keep their values consistent. This results in additional overhead and can increase the state space considerably.

6.1.4 Untyped Allocations

In C/C++ untyped allocations such as *malloc*, *calloc*, and *realloc* can easily be used to create overlays, which again require translation overhead to keep the corresponding non-overlaid objects consistent.

```
char* m = (char*) malloc(…);
…
A* pa = (A*)m;
B* pb = (B*)m;
```

Most coding guidelines for safety-critical software discourage the use of such dynamic memory allocations, particularly after initialization. Such memory allocations and the corresponding garbage collections can result in unpredictable behavior that could significantly affect both the performance and the verifiability of the code.

Applications that operate within a fixed, pre-allocated area of memory can avoid many of the problems associated with mishandling of memory allocations or de-allocations such as:

- Forgetting to free the memory
- Using memory after it was freed
- Attempting to allocate more memory than physically available
- Overstepping boundaries on allocated memory

These applications are much easier to verify for memory-related properties and to prove that they can always operate within the available resource bounds.

6.1.5 Unconditional Jumps

Jumps which are not strictly upwards in the block hierarchy can require extensive control-flow manipulation, including creation of redundant code, and should be avoided if possible. Such jumps add considerable modeling overhead for some model checkers. For example, *gotos* are not directly supported in JPF, but SPIN does handle them by default:

```
…
if (…) {
  …
  LABEL:
  …
} else {
  …
  goto LABEL;
  …
}
```

6.1.6 Multiple Inheritance

In C++, multiple-inheritance occurs when a class inherits from more than one parent. For example:

```
class B {
  …
  virtual void foo();
  …
};

class C : public B {
  …
  virtual void foo();
  …
};

class A : public B, public C
{
  …
};
```

Multiple inheritance used with several non-pure virtual bases cannot be translated into a simple class hierarchy in modeling languages such as Java, and might require significant delegation overhead to reach base class fields and virtual methods. The presence of multiple inheritance also requires explicit control about the constructor calling sequence, and therefore cannot be modeled with constructors in languages such as Java.

6.1.7 Use of `setjmp()` and `longjmp()`

In C/C++, `setjmp()` saves the contents of the registers at a particular state in the program and `longjmp()` will restore that state later. In this way, `longjmp()` "returns" to the state of the program when `setjmp()` was called.

In general these functions are problematic for C++ because they do not support C++ object semantics. In particular, they do *not* ensure the proper destructor calls and so objects may not be cleaned up properly, leaving the system in a bad state.

Using these functions may also degrade performance by preventing optimization on local variables. For C++ exception handling, try/catch constructs are recommended instead.

```
jmp_buf save;
...
int ret = setjmp(save);
if (ret == 0) {
  // first time action
  ...
  longjmp(save, SOME_ERROR);
  ...
} else if (ret == SOME_ERROR) {
  // jump back action
}
...
```

For model checking in particular, non-hierarchical long jumps are difficult to model with modeling languages like Java. These functions need to do register restoration which requires modeling of registers and their related operations, adding considerable overhead to model-checking effort.

6.1.8 Signal Handlers

In some operating systems, signal handlers are executed on a thread stack; they "overlay" the current thread execution (which may be used, for example, in combination with setjmp/longjmp to create user-domain lightweight threads). The thread model supported by modeling languages often cannot fully model signal handlers. This is certainly true in the case of a language like Java. Real-Time Specification for Java tries to address this to some extent, but it is not a complete solution.

6.1.9 Limit the Use of Pre-processor Directives

The C pre-processor is powerful, but unrestricted use of it can lead to code that is hard to understand and analyze. Limit its use to inclusion of header files and simple macro definitions. Avoid features such as token pasting, variable argument lists (ellipses), recursive macro calls, and macros expanded into incomplete syntactic units.

The use of conditional compilation directives and complex macro definitions must also be kept to a minimum. These make the job of modeling more complicated because they have to be translated into the modeling notations which often do not have similar built-in capabilities.

6.2 State-Reducing Design Measures

The measures listed in this section address reduction of the model checking relevant state space by dealing with a reduction of implementation-inflicted complexity.

6.2.1 Reduce Concurrency

From a model-checking perspective, the searched state space consists of all possible thread-state combinations, which implies that the level of concurrency has the biggest impact on state space size. As a consequence, reducing concurrency can be considered as the premier measure to ease model checking.

6.2.2 Number of Threads

Threads can be a useful abstraction and implementation mechanism to partition independent program actions. However, when there is coordination (or interference) between these threads, the required synchronization mechanisms increase the time, increase the state space, and introduce potential liveness and safety problems.

Some of these cases can be reduced with multiplexing patterns; for example, separating asynchronous event emitters from a synchronous event processor by means of an event queue mechanism—i.e. combining sequenced operations inside of one dedicated thread (the event processor) instead of using explicit synchronization between several threads to enforce the sequencing.

6.2.3 Number of Interleavings

Besides the raw number of threads, the state space is affected by the number of potential interleavings of these threads. While there exist automated techniques (partial-order reduction) to reduce these interleavings, most model checkers include some kind of interface to denote atomic sections (code which does not interfere with other threads). Previous versions of JPF supported two primitives: `Verify.beginAtomic()` and `Verify.endAtomic()`. They were used to mark a section of the code that would be executed atomically by the model checker so that no thread interleavings were allowed.

```
Verify.beginAtomic();

... // code without side effects outside this thread

Verify.endAtomic();
```

These calls are deprecated in the current version of JPF, which by default supports partial-order reduction instead. Use the `Verify.beginAtomic` and `Verify.endAtomic` primitives with care because a deadlock will occur when a thread becomes blocked inside of an atomic section.

6.2.4 Use Finite-State Space Abstractions

In order to successfully apply explicit-state model checking, defects must be detectable in a sufficiently small state space. This can be achieved either by means of heuristics that constrain the way the state space is searched, or by means of abstractions closing the state space (this was discussed in detail in Chapter 3). While the model checker can automatically apply certain abstractions, others should be used explicitly in the target program.

The most important state space reductions of the last category use specialized types to reflect value constraints. For example avoid using "int" and "long" types where the whole range of values is not needed. If a variable v can only take values -1, 0, or 1, then we can declare it as an enumerated type with only those three values. This helps the model checker limit the range of values that it needs to check for.

6.2.5 Counters

Counter variables are commonly used in many computer applications for different purposes. Here is a typical example where a variable is used to measures the progress of some activity of interest:

```
int progress, previous;
...
progress++;
...
if (previous < progress) {
  previous = progress;
  ...
}
```

If the exact value of the progress variable or the entire range of its values is not really required for checking the property of interest, then the use of this variable can cause unnecessary (potentially infinite) state space expansion. In this case, the progress variable can be reduced to a Boolean variable:

```
bool progress = false;
...
progress = true;
...
if (progress) {
  progress = false;
  ...
}
```

This modification immediately cuts down the number of states that need to be explored for this piece of code from $((2*\mathrm{MAXINT})+1)^2$ to only two possibilities: the progress variable taking only the value of true or false. Such small modifications can lead to exponential improvements in memory and time for a model checker.

6.2.6 Timers

While time values usually cannot be reduced in the target system, their usage can be encapsulated as an abstraction which can be replaced easily (e.g., by a non-deterministic choice) during model checking.

The following code shows two typical operations involving time implemented in C/C++: calculating a time delta between two execution points, and logging the current time at a specific point in the execution:

```
// time delta
struct timeval t1, t2;
struct timeval td, tdmax = { MAX_SEC, MAX_USEC };
...
gettimeofday(&t1, NULL);
...
gettimeofday(&t2, NULL);
timersub(&t2, &t1, &td);
if (timercmp(&td, &tdmax, >) {
  ...
}

...
// timed log
struct timeval now;
gettimeofday(&now, NULL);
log(&now, ...);
```

One way to dramatically reduce the state space of this code during model checking is to:

- Replace the time variables with non-deterministic choices (e.g., a simple integer whose value can be controlled by the model checker and possibly kept to a restricted range), and
- Replace all time comparisons with a simple non-deterministic Boolean choice.

The problem with doing this across the entire program is that it may involve modifications to many parts of the code—everywhere the corresponding variable has been used or referenced. This is a time-consuming task for realistic systems and is error-prone.

The alternative is to create an abstraction for the timer, a class called *StopWatch* which encapsulates all the relevant operations involving time. The code above can be reduced to the following:

```
// timed delta
StopWatch sw;

sw.start();
...
sw.stop();
if (sw.exceeded(MAX_MSEC)) {
  ...
}
// timed log
logTime();
log(...);
```

Now changes to the way time is modeled are all localized to the `StopWatch` class. You can switch between the real and modeled implementations with minimal effort, or even exclude timers from the model-checking process if necessary. As mentioned in the previous chapters, this approach also localizes the necessary instrumentation. For example, we only need to insert assertions in the methods provided by the `StopWatch` class rather than sprinkling them all across the target application.

6.2.7 Primitive Type Abstractions

An effective way to reduce the state space of a program is to replace the primitive types with the corresponding abstractions that encapsulate all the possible operations that are performed on these types.

These abstractions allow us to replace the actual primitive variables or the operations involving those with the non-deterministic choices with restricted range of values. This can cut down the number of states that has to be covered by the model checker dramatically.

As an example, consider the following class which encapsulates different operations that can be carried out on doubles.

```
class Double {
  double m_value;
public:
  Double () { m_value = NAN; }
  Double (double value) : m_value (value){}
  ...
  Double operator+ (double value) { ... }
  Double operator/ (double value) { ... }
  Double operator- (double value) { ... }
  Double operator* (double value) { ... }
  ...
};
```

Here is the implementation for the operator`==()` which implements the real behavior of the operator:

```
int operator==(double value) {
  return m_value == value;
}
```

With this abstraction it would be easy to reduce the potentially infinite state space required to model check the equality operator "==" with a binary choice (true or false)—just replace the "m_value == value" with "`chooseBoolean()`" in the body of the method. Appendix A shows the full C++ implementation of the `Double` class as an example.

The additional advantage of such abstractions is that they can provide convenient places for instrumentation. For example, in order to check whether a double number is valid and whether or not an operation results in overflow or underflow, instead of spreading the instrumentation all over the application code you can simply add the right instrumentation to the corresponding operator definition in this class, as shown below:

```
Double operator+ (double addend) {
  assert(!isnan(addend));
  assert(!isnan(m_value));
  assert(!isinf(addend));
  assert(!isinf(m_value));
  assert(...);
  ...
  return Double(m_value + addend);
}
```

6.3 General State-Relevant Design Principles

Even though they have less dramatic impact, there are a number of general object-oriented design principles which help to reduce model checker relevant state.

6.3.1 Avoid Redundancy

While not every form of redundancy is as bad from a verification perspective as it is from a maintenance point of view, behavioral redundancy to re-create (local) state can impose problems because the model checker does not distinguish between function local and object state. For example:

```
class A {
  void f (...) {
    X x(...);
    ... // do something with x w/o changing it
  }

  void g (...) {
    X x(...);
    ... // do something else with x w/o changing it
  }
};
```

We can change the above example by factoring out the declaration of `x` and turning it into object state.

```
class A {
  X x;
  ...
  void f(...) { /* use x */ }
  void g(...) { /* use x */ }
};
```

Another harmful example of redundancy can occur with redundant aggregates that could be turned into delegation objects. For example, consider the following example:

```
class A {
  X x;
  B b;
  ..
  void f (..) {
    x.modify();
    b.update(x);
  }
};
```

```
class B {
  X x;
  A a;
  ..
  void g (..) {
    x.modify();
    a.update(x);
  }
};
```

Classes A and B are creating instances of X, A, and B as data members. These redundant instances are usually a result of program extension—initially attempting to keep changes local (to a function member or class), but later realizing that the changes are required in a larger scope (a whole class or set of classes). We can modify this example by factorizing X, A, and B data members and turn them into delegation objects, as follows:

```
class A {
  X *x;
  void f (...) {
    x->modify();
    b.update(x); // update modified to take a reference
  }
};

class B {
  X *x;
  void g (...) {
    x->modify();
    a.update(x);
  }
};

...
X *x = new X(...);
A *a = new A(...);
B *b = new B(...);
a-> setX(x); b-> setX(x);
```

This assumes that A and B are used within the same thread; otherwise, you need to consider the additional synchronization overhead if X is shared between two different threads.

6.3.2 Use Polymorphism

Programs, especially those converted from non-OOP languages like C, sometimes use state where they should use inheritance. For example,

```
class A {
  int type;

  A (…) {
    type = TYPE_A;
    …
  }

  void foo(…) {
    …
    if (type == TYPE_A)
      … /* doAStuff */ ;
    else if (type == TYPE_B)
      … /* doBStuff */ ;
    …
  }
};
```

```
class B : public A {
  B (…) {
    type = TYPE_B;
    …
  }
  …
};
```

The variable `type` is used to explicitly store the type of an instance of class A or B. It is initialized in the constructors of both classes and is used in the `A::foo()` method to determine the exact type of the object so that the appropriate operations are carried out.

Besides being error prone (type initialization, branch completeness) and breaking fundamental object-oriented abstraction rules (base classes should not have to know about their concrete derived classes), this produces more code (the branches) and more data (type fields) that have to be handled by the model checker.

You can use the inheritance mechanism in C++ to do this more elegantly, as shown below:

```
class A {
  virtual void foo(…) {
    … /* doAStuff */
  }
};
```

```
class B : public A {
  virtual void foo(…)
{
    … /* doBStuff */
  }
};
```

We have observed this problem while model checking our SAFM case study, (section 9.4), which contained a number of bugs. The types of 21 derived classes were distinguished by looking at a string "name" data member which was declared in the base class and was set when the objects of those classes were instantiated. Every time it needed to perform one of two mutually exclusive operations based on the type of the object, a complex conditional statement involving object name comparisons was used to decide which one to perform.

We modified the code to use polymorphism as suggested in this guideline, which in turn reduced the complexity of the code. After the modification we model checked the code using JPF and noticed significant reduction in the number of byte codes executed, the memory consumed, and the execution time before it detected a given bug.

6.3.3 Leverage Model-Based Design

Model-based design provides useful hints of how a large system can be reduced so that its state space becomes searchable. If not inherently visible in the design (for example, by means of using a "State" design pattern), the model-relevant information should at least be included as comments.

6.3.4 Finite State Machines

A Finite State Machine (FSM) is one of the most suitable models for formal checks, especially for concurrent systems. However, FSMs can have problems with inheritance (the state model can change in derived classes) if state aspects are not factorized (e.g., with the State design pattern).

6.3.5 Use Platform-Independent Libraries and Abstraction Layers

Contemporary libraries can significantly exceed the size and complexity of applications. In practice, the application model or environment model will have to model some of these libraries (especially the ones without source code availability). We recommend the use of platform-independent libraries, such as POSIX (especially pthreads), to easily model complexity and increase opportunity to reuse models.

6.3.6 Limit the Scope of Data Declarations

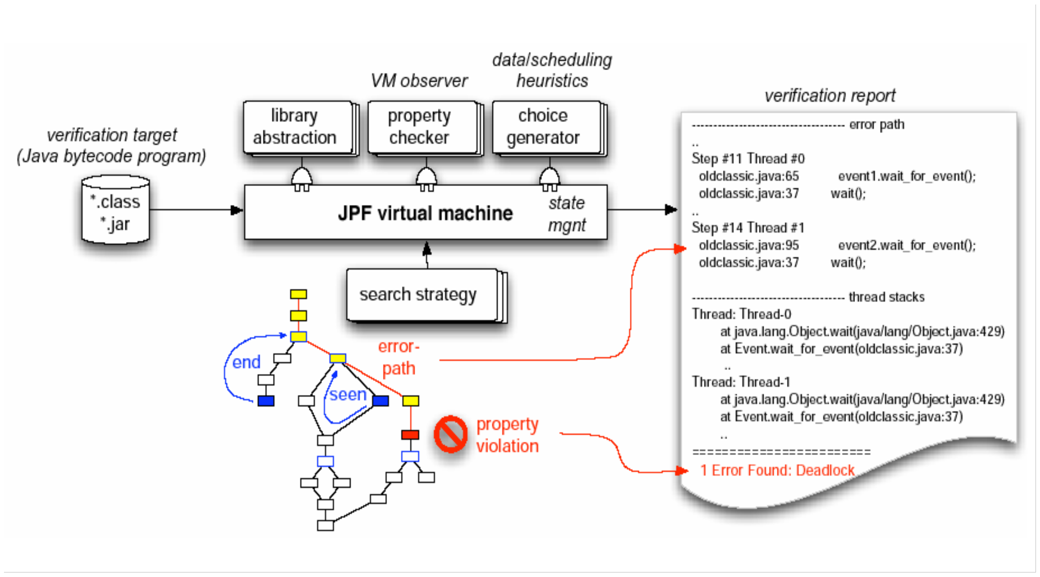
Limiting the scope of data declaration at the smallest possible level is consistent with the well known principle of data hiding. It stops one module from inadvertently referencing and modifying data values which are only meant to be used by another module. It also improves the clarity of the code.

Limiting data declaration scopes improves model checking because it reduces the number of data elements and statements that must be tracked during analysis in order to verify properties of interest.

7 Model Checkers

7.1 Java PathFinder

Java PathFinder (JPF) is a model checker that has been developed as a verification and testing environment for Java programs (Figure 5). It is available as open source at SourceForge.net (JPF website). It consists of a custom-made Java Virtual Machine (JVM) that interprets bytecode, combined with a search interface to allow the complete behavior of a Java program (including all interleavings of concurrent programs) to be analyzed. JPF itself is implemented in Java and its architecture is highly modular and extensible to support rapid prototyping of new features. JPF works at the bytecode level, and therefore it can uncover errors at both source and bytecode levels.



```
---------------------------------- error path
..
Step #11 Thread #0
  oldclassic.java:65        event1.wait_for_event();
  oldclassic.java:37       wait();
..
Step #14 Thread #1
  oldclassic.java:95        event2.wait_for_event();
  oldclassic.java:37       wait();

---------------------------------- thread stacks
Thread: Thread-0
        at java.lang.Object.wait(java/lang/Object.java:429)
        at Event.wait_for_event(oldclassic.java:37)
         ..
Thread: Thread-1
        at java.lang.Object.wait(java/lang/Object.java:429)
        at Event.wait_for_event(oldclassic.java:37)
        ..
========================
 1 Error Found: Deadlock
```

Figure 5 Java PathFinder Architecture

7.1.1 Property Specifications

The most straightforward way to specify and check simple safety properties in JPF is to use Java assertions inside the application under analysis. This allows the specification of properties that only depend on the application data values (e.g., parameter value intervals). Violations are caught by JPF. The drawbacks of this method are that it requires access to the application sources, and that it can significantly increase the state space if the property requires evaluation state itself (e.g., for properties implemented as automatons).

The other way to specify properties is by using `gov.nasa.jpf.Property` or `gov.nasa.jpf.GenericProperty` instances to encapsulate property checks.

```
public interface Property extends Printable {
boolean check (Search search, VM vm);
String getErrorMessage();
}
```

The user typically creates an instance of such a class and provides an implementation for its `check()` method which does the main work for checking the property. The `check()` method is evaluated after each transition. If it returns false and termination has been requested, the search process ends, and all violated properties are printed (which potentially includes error traces).

JPF comes with generic Property classes for the following properties:

- No Deadlocks
- No Assertion Violation
- No Uncaught Exceptions (i.e., not handled inside the application)

Another way of specifying properties is through the use of two listener classes: `gov.nasa.jpf.SearchListener` and `gov.nasa.jpf.VMListener`. The listeners can subscribe to events during the search, making JPF easily extensible. They can be used to implement more complex checks that require more information than what is available after a transition is executed. The rich set of callbacks enables listeners to monitor almost all JPF operations and translate them into internal state.

Property objects can be configured statically or dynamically through a programmatic API. To activate checking for these properties, add the corresponding class names to the colon-separated list of class names specified under `search.properties` in a JPF configuration file:

```
search.properties=\
    gov.nasa.jpf.jvm.NotDeadlockedProperty:\
    gov.nasa.jpf.jvm.NoAssertionViolatedProperty:\
    gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty:\
    x.y.z.MyNewProperty
```

Although JPF previously supported LTL checking, this feature is no longer supported.

7.1.2 Environment Modeling

In JPF, Java class files can be processed in two different ways:

- As ordinary Java classes managed and executed by the host JVM (e.g., standard Java library classes, JPF implementation classes)
- As "modeled" classes managed and processed (verified) by JPF

We have to clearly distinguish between these two modes. In particular, JPF's "Model" layer has its own class and object model, which is completely different than and incompatible with the hidden class and object models of the underlying host JVM executing JPF.

Each standard JVM supports a Java Native Interface (JNI), that is used to delegate execution from the JVM-controlled bytecode down into the platform-dependent native layer (machine code). This is normally used to interface certain functionalities such as I/O or graphics to the platform OS and architecture. JPF provides an analogous mechanism to lower the "execution" level in JPF from JPF-controlled bytecode into JVM-controlled bytecode. This mechanism is called Model Java Interface (MJI). It supports the creation of dedicated classes to be executed by the underlying JVM rather than JPF. Such classes are not model checked by JPF.

MJI offers a wide range of applications. These include the following:

- Interception of native methods including calls to native libraries: We can easily model and abstract the operation of these libraries.
- Interfacing of JPF system-level functionality: Some system-level functions of standard library classes (especially `java.lang.Class` and `java.lang.Thread`) must be intercepted even if they are not native, because they have to affect the JPF internal class, object, and thread model.
- Extending the functionality of JPF without changing its implementation.
- State space reduction: By delegating bytecode execution into the non-state-tracked host JVM, we can cut off large parts of the state space. This improves the efficiency and scalability of the analysis.
- Collecting information about JPF's state space exploration.

Software model checking is all about making the right choices to guide the analysis tool (in this case, JPF), to systematically explore interesting system states within the resource constraints of the tool and execution environment. Many of these choices may be non-deterministic—for example, what thread to schedule next or what the value is of the next floating point number that is entered as an input from the environment.

Support for simple "random" data acquisition (using the `gov.nasa.jpf.jvm.Verify` interface) has been included in JPF since the beginning. For example, `Verify.getInt(2)` will non-deterministically return a value in the range 0-2, inclusive, which the model checker can then trap during execution. JPF tries all the possible values systematically. For more complex choices however (e.g., values for a data type with infinite set of values such as floating point numbers) a more powerful mechanism is required. To do this, JPF uses *Choice Generators* which allow us to use heuristics to make the set of choices finite and manageable. For example, you can choose to use the set of all floating point numbers within a given boundary, each separated by a delta value from the next.

This mechanism is completely configurable and is decoupled from the rest of the system, allowing a lot of flexibility in guiding the search. Choice generators are the preferred mechanism for modeling the environment and can be implemented using MJI. This allows choice generators to act as test drivers for the model checker.

7.1.3 State Space Reduction

JPF is a so-called explicit-state model checker, since it enumerates all visited states, and therefore suffers from the state explosion problem inherent in analyzing large programs. It also contains garbage collection, since a typical Java program will be infinite state (state size will grow infinitely) without garbage collection.

Previous versions of JPF used a facility for marking blocks of code as atomic, using calls to `Verify.beginAtomic()` and `Verify.endAtomic()`. This is not supported in the latest version of the tool. Instead, it supports dynamic partial-order reduction by default, as a means of reducing the number of interleavings analyzed by the tool while checking concurrent code.

7.1.4 Search and Partial Coverage

JPF supports well-known search strategies such as BFS and DFS as well as various heuristic-based searches such as A*, Best-First, and Beam Search. You can sum a number of heuristics during the search, and set search depth as well as the number of elements in the priority queue associated with searches such as simple BFS or the heuristic searches described above. Dynamic annotations in the source code can cause the model checker to increase or decrease heuristic values. Previous versions of JPF supported the calculation of structural coverage (branch and statement coverage) during model checking; coverage could also guide heuristic search. The current version of the tool however does not support these features. There are plans to include such facilities in the tool in the future.

7.1.5 Scalability

Scalability of program model checkers such as JPF encompasses two aspects:

- How large a program can be model checked, and
- Once a defect has been detected, how readily meaningful debugging information can be derived from trace.

How large a program JPF can model check is highly dependent on the structure and amount of nondeterminism in the application. For example, UI applications or certain Web applications which have a central control loop (i.e., reactive state machines) that reacts to user input or signals from the environment are amenable to model checking. We have applied JPF successfully to multithreaded Mars Rover software which was around ten thousand lines of code (10 KLOC.) We have also applied JPF successfully to multithreaded UI programs with 33 KLOC of application source plus many more lines of code in the libraries. At this point, the largest program checked by JPF is a commercial Web application of several hundred thousand lines of code, in which JPF found a deadlock involving about 20 threads; JPF executed more than 200 million byte code instructions of the target application in about 4 hours. Chapter 6 in this guidebook discusses ways to design applications to mitigate the scalability problem.

The second aspect of scalability is the ability to derive meaningful debugging information from an execution trace that manifests a defect. This sense of scalability can represent a severe limitation if the defect occurs long into the program execution. JPF provides limited capabilities here—for example, JPF's DeadlockAnalyzer and ChoiceTracker, which are used in user interface applications to generate trace reports showing only user-input events. They are used also in analyzing Unified Modeling Language (UML) to limit trace reports to showing only state-machine environment events.

7.1.6 Other Features

JPF includes the capability to perform symbolic execution of Java programs, including complex data such as linked lists and trees. See section8.3.3 for details.

7.2 SPIN

SPIN (Holzmann 2004), winner of the 2001 ACM Software Systems Award, is one of the leading model checkers used for analyzing the logical consistency of concurrent and distributed systems, specifically of data communication protocols. Annual SPIN workshops have been held since 1995. The tool's main focus is on proving the correctness of process interactions and not the internal computations of the processes. Processes represent system components that communicate with each other.

7.2.1 Modeling Language

The modeling language used for SPIN is called Promela (*Pro*cess *Me*ta *La*nguage). In fact, the name SPIN stands for Simple Promela Interpreter. Promela is a powerful C-like specification language with a variety of synchronization primitives that enables construction of models of distributed systems. Promela is a guarded command language for specifying possibly non-deterministic system behaviors in a distributed system design. It allows for the dynamic creation of concurrent processes which communicate synchronously or asynchronously via message channels.

For example, the following Promela specification describes two processes *P* and *Q* that synchronize over a critical region.

```
active proctype P()
  { do::!crit[1] ->
      { crit[0] = true
        /* do critical section here … */
        ; crit[0] = false }
    od  }

active proctype Q()
  { do::!crit[0] ->
      { crit[1] = true
        /* critical section */
        ; crit[1] = false }
    od }
```

A Promela model consists of declarations of types, channels, variables, and processes, as well as facilities for initializing the processes. For example,

```
mtype = {MSG, ACK};
chan a,b,c = ...
bool flag

proctype Sender() { ... }

proctype Receiver() { ... }

init { ... }
```

A model consists of a set of concurrent processes, each in turn consisting of a set of atomic guarded actions or transitions. An action whose guard is true is called enabled and can be executed; otherwise, it is disabled and cannot be executed. Actions are not really executed concurrently; rather, they are non-deterministically interleaved, and this is used to model concurrency. Also processes do not have to terminate, which is useful when modeling reactive systems.

In addition to the usual primitive data types found in other languages like C/C++ (e.g., bool, byte, int, and short) Promela support data types to represent message types and channels. It does not support types such as reals, floats, or pointers. This is deliberate because verification models are meant to model coordination and not computation. Promela also supports composite types such as records and arrays.

7.2.2 Embedding C Code

Promela supports the use of embedded C code fragments inside the models through the use of the `c_code` primitive.

```
c_code <optional logical expression in C> {
     <C code fragment>
}
```

For example:

```
c_code [foo->ptr != 0 && step.i < 5 && step.i >= 0]
{
      foo->ptr.x[step.i] = 12;
}
```

If the logical expression is specified, its value is evaluated *before* the `c_code` fragment inside the curly braces is executed. If the result of the evaluation is non-zero, the `c_code` fragment is executed. If the result of the evaluation is zero, the code between the curly braces is ignored, and the statement is treated as an assertion violation. The example above adds checks for null pointers and bounds on array indices.

Many other code verifiers which do not necessarily use Promela as their modeling language use SPIN as a back-end verifier.

7.2.3 State-space Reduction

Two language statements are used to reduce the number of states in a Promela model: *atomic* and *d-step*.

```
atomic { statement1; ...; statementn }
d-step { statement1; ...; statementn }
```

Atomic is used to group statements of a particular process into one atomic sequence. The statements are executed in a single step and are not interleaved with statements of other processes. The atomicity is broken if any of the statements is blocking. That is, statements of other processes can be interleaved in between.

d-step can also be used to execute a number of statements in one step. The difference is that no intermediate states are generated or stored. If one of the statements blocks, this is considered a runtime error.

7.2.4 Property Specifications

Users can specify assertions using the `assert(expr)` statements. An assert statement is used to check if the property specified by the expression expr is valid within a state. If `expr` evaluates to 0, this implies that it is not valid and SPIN will exit with an error. The user can also use LTL formulas to specify general correctness requirements (e.g., temporal properties). See section 2.3.3.2 for a more detailed explanation.

7.2.5 Search and Coverage

SPIN has a highly optimized state exploration algorithm. It supports random, interactive and guided simulation, and both exhaustive and partial coverage, based on either *depth-first* or *breadth-first* search.

To reduce the problem of running out of memory storing the states that have been visited (so that the search will not explore them again), SPIN provides an option called *bitstate hashing*, also known as *supertrace*. Bitstate hashing uses memory as a bit vector. When a state is visited during SPIN's exploration, its state vector is hashed to a bit in memory: the bit indicates whether the state has been visited previously. If a different state vector hashes to the same bit location, SPIN will erroneously conclude that the state has been visited and will terminate exploration from that state. This may lead to false negatives (errors may not be found), but not false positives (no false error reports).

Paradoxically, using the bitstate option may lead to increased coverage of the state space, because more of it can be explored in the given amount of memory.

7.3 Bandera

The goal of the Bandera project is to integrate existing programming language processing techniques with newly developed techniques to provide automated support for the extraction of safe, compact, finite-state models that are suitable for verification from Java source code. While the ultimate goal is fully automated model extraction for a broad class of software systems, Bandera takes as a given that guidance from a software analyst may be required.

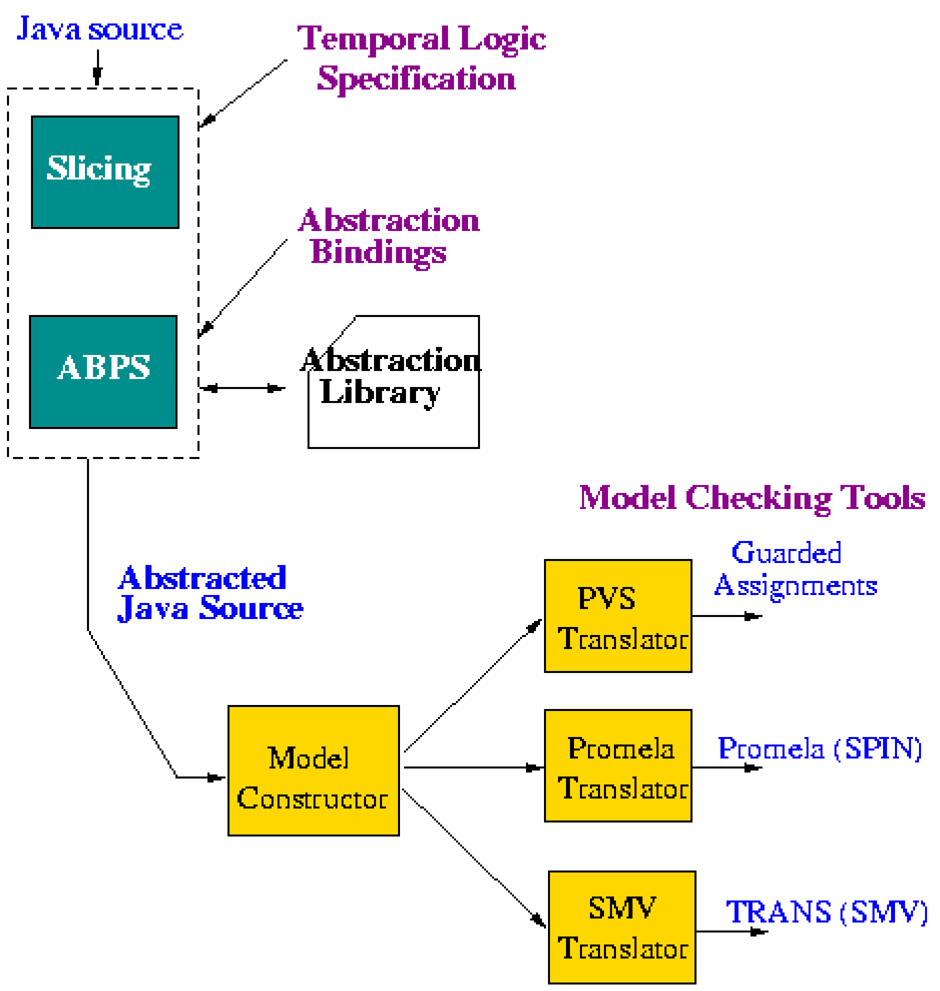


Figure 6 The Bandera Toolset

As illustrated in Figure 6, the Bandera toolset is designed to be an open architecture in which a variety of analysis and transformation components may be incorporated.

7.4 BLAST

The Berkeley Lazy Abstraction Software Verification Tool (BLAST) is a software model checker for C programs. The goal of BLAST (BLAST website) is to be able to check that software satisfies behavioral properties of the interfaces it uses. Blast uses counterexample-driven automatic abstraction refinement to construct an abstract model which is model checked for safety properties. The abstraction is constructed on the fly, and only to the required precision.

7.5 Bogor

Bogor (Robby, Dwyer, and Hatcliff 2006) is an extensible software model-checking framework which includes:

- Software model checking algorithms
- Visualizations
- A user interface designed to support both general-purpose and domain-specific software model checking

Bogor provides support for object-oriented features such as dynamic creation of threads and objects, object inheritance, virtual methods, exceptions, and garbage collection. Bogor can be extended with new modeling types, expressions, and commands, new algorithms (e.g., for state-space exploration, state storage, etc) and new optimizations (e.g., heuristic search strategies, domain-specific scheduling, etc.).

7.6 BOOP Toolkit

The BOOP Toolkit has been developed at the Institute for Software Technology at Graz University of Technology. It is based on the SLAM project and uses the same main concept of verification by abstraction and refinement to determine the reachability of program points in a C program. The BOOP Toolkit follows three phases:

1. *Abstraction* using theorem proving and predicate abstraction
2. *Model Checking* of the abstract program to search for erroneous execution traces (with respect to a specification that must be provided by the user)
3. *Refinement*. If the model checker found an error path that is feasible in the original C program, it is reported to the user. If the path is not feasible, the abstraction is refined by adding more detail and the process is reiterated.

The toolkit is downloadable from Source Forge (BOOP website).

7.7 LURCH

LURCH (Menzies *et al*. 2004) uses random search to explore a state machine's state space (see section 5.1.5.2). Because the search is random, it gives no guarantee that the state space has been exhaustively explored, so LURCH cannot be used for verification. However, any errors it finds are real errors, so it can be used for debugging. It can find assertion violations, deadlock, and cycle-based errors (no-progress cycles, and violations of temporal logic properties specified as acceptance cycles); finding cycle-based errors requires a state hash table. LURCH's input format is concurrent finite-state machines, where transitions may be specified by C code.

In a case study of running model checkers to find seeded errors in a flight guidance system model, NuSMV (a model checker not described in this report) had a mean runtime of 2 hours where LURCH had a mean runtime of 3 minutes, yet LURCH found

40 of the 44 seeded errors that NuSMV found. In other demonstration problems, LURCH found all the errors that SPIN found when SPIN could terminate with the given resources. LURCH is claimed to scale to larger problems, and tends to use a stable amount of resources.

7.8 MCP

Model Checker for C++ (MCP) is an explicit-state software model checker being developed by the Robust Software Engineering group at NASA Ames Research Center (Thompson and Brat, 2008). MCP was constructed specifically to allow programs written in C or C++ to be model-checked directly, without requiring prior translation or model extraction. It builds upon the LLVM (low level virtual machine) compiler infrastructure (LLVM website), thereby avoiding the requirement to directly recognize the extremely complex C++ language at source-level. This approach allows MCP to support the entire C++ language, including templates. The C language is handled fully, not just as an improper subset of C++. Experience from the JPF project has demonstrated the utility of program model checking. However, current flight software is mostly implemented in C, not Java, and in the future it seems increasingly likely that C++ will become the platform of choice. The MCP model checker was developed to fill the requirement for an explicit-state software model checker, in the style of JPF, that fully supports the C++ language. As a test example, it was applied to the unmodified C code of the prototype CEV Onboard Abort Executive Flight Rules Checker.

7.9 SLAM

SLAM is a Microsoft project that blurs the line between static analysis and model checking and deductive reasoning. The main goal of SLAM is to check temporal safety properties of C programs (it actually checks that a program correctly uses the interface to an external library) while minimizing the problem of false positives (by using counterexample-driven refinement) without overburdening the users with code annotations. SLAM is original in the sense that it first creates a Boolean abstraction of the original program, then refines the abstraction until it can prove the property or produce a counterexample.

SLAM is one of the rare model checkers with a successful technology transfer story. Indeed, the SLAM analysis engine forms the core of a newly released tool called Static Driver Verifier (SDV) that systematically analyzes the source code of Microsoft Windows device drivers against a set of rules that define what it means for a device driver to properly interact with the Windows operating system kernel.

7.10 VeriSoft

VeriSoft (Godefroid, 2005) is a model checker for C and C++; other languages can be used, but components in other languages are treated as black boxes. VeriSoft has been used to find defects in very large telecommunications programs (Chandra, Godefroid, and Palm 2002). It is available for download at the VeriSoft website. This package includes a version of VeriSoft for analyzing multiprocess C or C++ programs whose processes communicate via a predefined set of types of communication objects. It is a "stateless" model checker, in that visited states are not saved. It uses clever partial-order reduction search algorithms utilizing program analysis to avoid revisiting states, and guarantees complete coverage of the state space to some depth, while preserving correctness properties.

Verisoft can search for four basic types of errors:

- Deadlocks.
- Divergences. A divergence occurs when a process does not attempt to communicate with the rest of the system for more than a given (user-specified) amount of time.
- Livelocks. A livelock occurs when a process is blocked during a sequence of more than a given (user-specified) number of successive states in the state space.
- Violations of state assertions, the assertions having been stated using a special operation `VS_assert(boolean_expr)`.

To represent non-determinism in the model or environment, VeriSoft provides a special operation `VS_toss` to express non-deterministic choice, which is like `Verify.random` in Java PathFinder.

8 Automated Testing

8.1 Introduction

Software testing is the most commonly used technique for verifying and validating software. While testing may not provide the same rigor as model checking, it is far more scalable and usually does not require a significant tool infrastructure; thus its *initial* cost and learning curve is low. However, for complex systems these benefits are more than offset by the fact that it is hard to develop the "right" test cases that provide sufficient coverage. This difficulty often reduces coverage despite use of many test cases. The consequence is that testing is a very labor intensive process that typically accounts for about half the total cost of software development and maintenance while yielding poor reliability due to inadequate coverage (Bezier 1990)].

Automating testing not only reduces the cost of verifying and validating software but also increases its reliability. A recent report by the National Institute of Standards and Technology estimates that software failures currently cost the US economy about $60 billion every year, and that improvements in software testing infrastructure might save one-third of this cost (NITT 2002). Testing, and in particular test case generation, lends itself to automation and has been the focus of much research attention; it has also been adopted in industry (Parasoft; T-VEC; Drusinsky 2000; Grieskamp *et al.* 2002). We distinguish between two main approaches to automatic test case generation: *black box* and *white box*. In the black-box approach, test cases are generated based on a specification of the inputs, while in the white-box approach, test cases are generated based on the analysis of the software under test or of a model of this software.

The quality of a test suite is measured in terms of test *coverage*, such as structural code coverage, MC/DC, path coverage, and requirements coverage. In addition to program inputs, test cases also need to specify the expected output. This can be done by using *testing oracles* that encode the functional specifications of the software under test as assertions in the code, post-conditions, etc.

In the remainder of this section, we discuss approaches to black-box and white-box testing that combine *symbolic execution* and model checking techniques for verification of Java programs. These approaches have been implemented in the Java PathFinder model checking tool. The originality of these techniques is that they handle complex data structures and multithreading, and allow for flexible coverage criteria.

8.2 Black-Box Testing

Black-box testing assumes an external perspective of the system under test (SUT). It uses concrete input test data to obtain results from the SUT, which are then checked against some oracle. Usually, the oracle uses functional properties, comparing the output data with expected values. The approach is easy in general, but quickly becomes problematic for complex SUTs, especially if one of the following conditions exists:

a. The set of possible, relevant inputs is large (this can be aggravated by particular implementations of algorithms that lead to numeric instability; i.e., the data might not seem to be suspicious from a domain perspective but still causes failure in a given implementation).

b. The SUT behavior cannot be completely controlled by input data (for example, its behavior depends on scheduling in concurrent programs).

c. The result data are complex and requires expensive analysis to verify.

d. The result data can be subject to "accidental correctness"; i.e., the SUT produces a right result for wrong reasons. This usually is caused by a small set of possible outputs, computed from a large input space.

e. Running the test requires manual action to enter inputs, start the SUT, or analyze the output.

f. There is no measure of relevant test coverage.

None of these problems can be overcome by strict black-box testing, but other V&V methods and tools can be used to mitigate them:

1. Model checking can produce minimal sets of test cases that guarantee certain coverage criteria for example by using symbolic execution modes of the model checker (mitigates conditions *a* and *f*). This is subject of a more detailed discussion in section 8.3, "White-Box Testing."

2. Dedicated test systems providing test harness infrastructure can be used to automate test execution (for example, for regression tests, mitigating *e*); this is discussed in the next section.

3. Dedicated verification runtime environments (e.g., virtual machines) can be used to simulate the program environment that cannot be controlled from a test application. These runtimes can also perform complex non-functional checks and defect analysis (mitigating *b* and *c*); this is discussed in the next section.

4. Code annotations can be used in combination with specific virtual machines (VMs) or code generators to automatically create and run test cases (which saves the effort of manually creating test harnesses and/or explicit code instrumentation). This is discussed in the next section. This is especially helpful for achieving the "test-like-you-fly/fly-like-you-test" goal. Verification-oriented runtime systems (like a program model checker) also enable checking non-functional properties that would otherwise not be amenable to testing (e.g., "no memory allocation in function x" or "function y has to be reentrant"). This mitigates *c*, *d*, and *e*.

5. Contracts in the code can be used to specify fine-grained properties, which reduces the black box property to check to a generic[1] "don't violate any

[1] See section 2.3.1.1* for the definition of "generic" properties.

contracts" (this is especially helpful to avoid "accidental correctness"). This is discussed in section 2.3.4.1. This can also be achieved by aspect-oriented programming, but at the cost of keeping the aspect system in sync with the SUT sources.

8.2.1 Dedicated Test Harness Frameworks

The most prominent example of a test framework is the JUnit system of Kent Beck and Erich Gamma (JUnit website). JUnit targets Java, but has spawned a large number of implementations for other languages (CPPUnit for C++, fUnit for Fortran, NUnit for C#, PyUnit for Python, etc.).

JUnit's main purpose is to provide infrastructure for efficient creation and execution of regression tests. This is achieved by creating JUnit-specific test case modules (classes), which call SUT functions and use the JUnit library to implement `property checks:import sut.MySUT;`.

```
import org.junit.*;
import static org.junit.Assert.*;

class MySUT_Test {
  //.. set up static environment
  static MySUT sut = …

  @Before
  void reset() {..}

  @Test
  void test_foo() {
    sut.foo();
    assertEquals(sut.getResult(), 42.0); // JUnit property
implementation
  }

  @Test(expected=ArithmeticException.class)
  void test_bar() {
    sut.bar(-1);
  }
  …
}
```

JUnit is meant to be used from inside an integrated development environment (IDE) (such as Eclipse, NetBeans, etc.), but test cases can also be executed from the command line by means of a runtime system that loads the test module, looks for JUnit-specific markups (e.g., the @Test annotation), automatically calls the associated methods (e.g., `test_foo()`), collects and checks output (e.g., with `Assert.assertEquals()`), and finally keeps statistics over all executed tests:

```
> java –ea org.junit.runner.JUnitCore MySUT_Test
```

Note that newer versions of JUnit do support a generic exception property (like `"@Test(expected=ArithmeticException.class)"`), but this depends on Java-specific runtime features. In general, the amount of code required to set up and execute test cases can be quite substantial, which is mostly due to missing support for input variation and the lack of generic properties beyond exceptions.

It should be noted that both limitations can be overcome by running JPF from inside a JUnit test method; i.e., by executing the SUT by JPF, and then using a generic "no property violations detected by JPF" property as the JUnit test assertion. In this case, both input data variation and more generic property checks (e.g., "no race condition") can be delegated to JPF. This approach is used for JPF's regression test suite itself, but is of course Java specific.

Since black box testing does not use any insights into the SUT, automating tests with systems like JUnit should be accompanied by measuring code coverage of the tests, either by using compiler options and tools for statically compiled languages (like gcov for C/C++), runtime monitors, or special VMs like JPF. Basic function and (binary) instruction coverage provides a good start, but should be augmented by branch coverage where possible (JPF provides a mode to calculate bytecode, extended basic block, and branch coverage). Tests without coverage information bear little verification credit, since—depending on input sets and runtime environments—there is no indication of how much SUT behavior is exercised by the test cases.

8.2.2 Test-Related Code Annotations and Runtime Systems

As useful as test systems like JUnit are, test cases do require a significant amount of work to create and maintain, especially since they are implemented outside of the SUT sources, and need to be kept up to date with SUT changes.

It is therefore desirable to have an annotation system for in-source specification of tests; that is, to keep test cases next to the SUT functions they refer to. However, such an annotation system has to meet several requirements in order to be effective and gain acceptance:

- The test specification format must be easy to read, preferably using target language expressions (like `"foo(20) < 42"`).
- The format must be concise enough to avoid cluttering SUT sources with test annotations (for example, not greater than 20% of the non test related lines of code), which especially requires a notation for input data variation.
- The annotation cannot produce code that might interfere with the SUT, otherwise it would violate the "test-what-you-fly/fly-what-you-test" principle.
- Test property types should be extensible and configurable, depending on which runtime execution system is used.

Since such a system is meant to be the first line of verification efforts (particularly for low-level function tests), its most important aspect is that it actually makes life easier for the developer, mitigating the burden of creating and maintaining a separate body of test code.

The JPF distribution includes a test annotation system that is based on Java annotations (http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html), and can be used with a minimal stand-alone execution runtime, or with JPF itself (in order to verify complex properties).

```
class MySUT {
  …
  @Test({"(42) == 43",
         "(-1) throws IllegalArgumentException"})
  int foo (int a) {
    if (a < 0)
      throw new IllegalArgumentException("negative values not
supported");
    return a+1;
  }

  @Test("this(4[23]).('bla') matches '4[23]BLA' ")
  String baz (String s){
    return (id + s.toUpperCase());
  }

  @Test("(1.2) satisfies NoAlloc"
  void bar (double d){
    …;
  }
```

The first `@Test` annotation specifies two tests with different inputs and goals (`"new MySUT().foo(42) == 43"` and `"new MySUT().foo(-1) throws IllegalArgumentException"`).

The second `@Test` varies input data and matches the results against a regular expression (`"new MySUT(42).baz('bla')"` and `"new MySUT(43).baz('bla')"` match the regular expression `"4[23]BLA"`).

Those two test specs can be executed and evaluated with a simple, stand-alone runtime environment that uses Java reflection to read annotations and call the appropriate constructors and methods. The JPF *MethodTester* application is an example of such a runtime; here is the output of running it on the example shown above.

```
> java gov.nasa.jpf.MethodTester MySUT
@ MethodTester - run @Test expressions, v1.0
@ target class:  MySUT
@ ==================================== tests:
@ ---- testing method: int MySUT.foo(int)
@ test spec: "(42) == 43"
@
@ goal: "==" {43}
@ execute: MySUT.foo(42)
@ returns: 43
@ Ok

@ test spec: "(-1) throws IllegalArgumentException"
@
@ goal: throws IllegalArgumentException
@ execute: MySUT().foo(-1)
@ throws: java.lang.IllegalArgumentException: negative values not
supported
@ Ok
…
```

The third `@Test` example requires a more sophisticated runtime environment to verify that there are no object allocations during the execution of method bar(). JPF itself provides such an environment, and consequently can be used to execute the MethodTester itself (i.e., it becomes the VM running the MethodTester application, which runs the SUT).

```
> jpf gov.nasa.jpf.tools.MethodTester MySUT
…
@ ---- testing method: void bar(double)
@ test spec: "(1.2) satisfies NoAlloc"
@
@ goal: alloc limit <= 0
@ execute: MySUT.bar(1.2)
@ FAILURE
```

To further automate this process, the JPF execution in turn can be part of a JUnit test suite that includes annotation-based entry-level method tests and complex unit and integration tests requiring separate test harnesses.

Again, the recursive application of runtime environments to add additional testing features and property types depends on Java (or an equivalent VM-based programming environment). For other languages (like C or C++), annotations have to be implemented as special comments that are processed by a source-to-source compiler that generates test sources. This is in general less convenient and usually does not allow generic properties (like "no race conditions", or "no allocations"), but still can be considered as an effective first-line defense to reduce testing costs.

8.3 White-Box Testing

We discuss here a framework that combines symbolic execution and model-checking techniques for automated test case generation in the context of Java programs. The framework is typically used for test input generation for *white-box* testing. The framework model checks the program under test. Test case generation is guided by a testing coverage criterion, e.g., branch coverage. The model checker explores systematically the paths of the program under test and records the paths that satisfy the coverage criterion. Symbolic execution, which is performed during model checking, computes a representation, i.e., a set of constraints, of all the inputs that execute those paths. The actual testing requires solving the input constraints in order to produce concrete test inputs that can be executed.

8.3.1 Symbolic Execution

We first provide some background on symbolic execution. The main idea behind symbolic execution (King 1976; Clarke 1976) is to use symbolic values, instead of actual data, as input values and to represent the values of program variables as symbolic expressions. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs. The state of a symbolically executed program includes the (symbolic) values of program variables, a path condition (PC), and a program counter. The path condition is a (quantifier free) boolean formula over the symbolic inputs: it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular associated path. The program counter defines the next statement to be executed. A symbolic execution tree characterizes the execution paths followed during the symbolic execution of a program. The nodes represent program states and the arcs represent transitions between states.

Consider the code fragment below, which swaps the values of integer variables `x` and `y`, when `x` is greater than `y`.

```
int x, y;
1:if (x > y) {
2:  x = x + y;
3:  y = x - y;
4:  x = x - y;
5:  if (x > y)
6:    assert (false);
}
```

Figure 7 shows the corresponding symbolic execution tree. Initially, PC is true and `x` and `y` have symbolic values $\mathtt{SYM}_X$ and $\mathtt{SYM}_Y$, respectively. At each branch point, PC is updated with conditions on the inputs that guarantee execution on alternative paths. For example, after the execution of the first statement, both `then` and `else` alternatives of the `if` statement are possible, and PC is updated accordingly. For the `then` branch, PC is updated with $\mathtt{SYM}_X > \mathtt{SYM}_Y$ (which makes the branch condition true), while for the `else` branch, PC is updated with $\mathtt{SYM}_X \bullet \mathtt{SYM}_Y$ (corresponding to the branch condition being false). If the path condition becomes unsatisfiable, i.e., there is no set of concrete inputs that satisfy it, this means that the symbolic state is not reachable, and symbolic execution does not continue for that path. For example, the assertion at line (6) is unreachable.

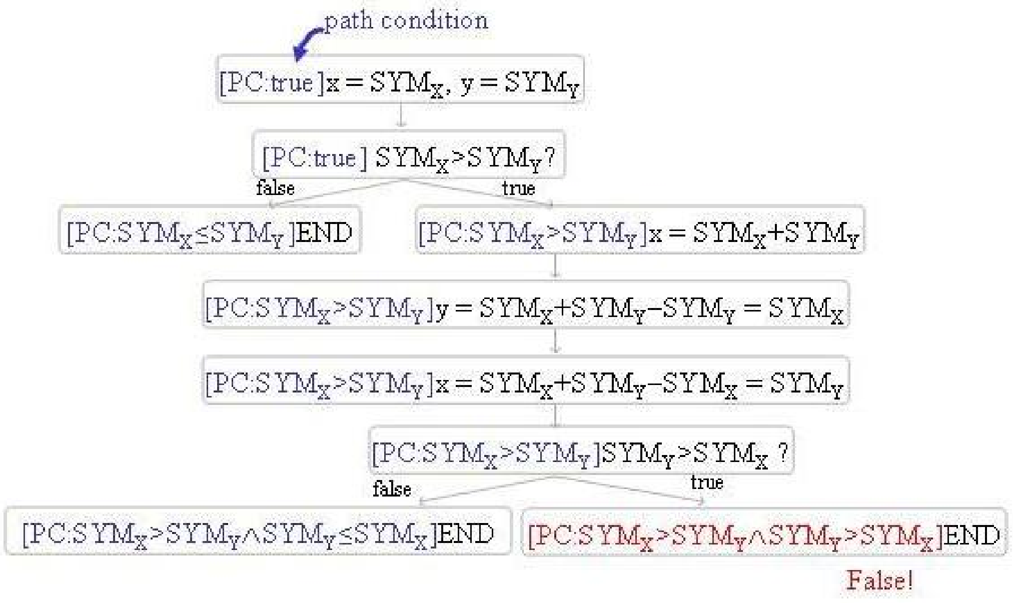

path condition
[PC:true]x = SYM_X, y = SYM_Y
[PC:true] SYM_X>SYM_Y?
false
true
[PC:SYM_X≤SYM_Y]END
[PC:SYM_X>SYM_Y]x = SYM_X+SYM_Y
[PC:SYM_X>SYM_Y]y = SYM_X+SYM_Y−SYM_Y = SYM_X
[PC:SYM_X>SYM_Y]x = SYM_X+SYM_Y−SYM_X = SYM_Y
[PC:SYM_X>SYM_Y]SYM_Y>SYM_X ?
false
true
[PC:SYM_X>SYM_Y∧SYM_Y≤SYM_X]END
[PC:SYM_X>SYM_Y∧SYM_Y>SYM_X]END
False!

Figure 7 Symbolic Execution Tree

8.3.2 Generalized Symbolic Execution

Using symbolic execution for test input generation is a well-known approach, but typically only handles sequential code with simple data. In (Khurshid, Pasareanu, and Visser 2003; Pasareanu and Visser 2004), this technique has been extended to support advanced constructs of modern programming languages, such as Java and C++. The approach handles dynamically allocated structures, arrays, as well as concurrency.

The algorithm starts execution of a method on inputs with uninitialized fields and uses lazy initialization to assign values to these fields; i.e., it initializes fields when they are first accessed during the method's symbolic execution. This allows symbolic execution of methods without requiring an a priori bound on the number of input objects. When the execution accesses an uninitialized reference field, the algorithm non-deterministically initializes the field to null, to a reference to a new object with uninitialized fields, or to a reference of an object created during a prior field initialization; this systematically treats aliasing. When the execution accesses an uninitialized primitive (or string) field, the algorithm first initializes the field to a new symbolic value of the appropriate type and then the execution proceeds.

When a branching condition on primitive fields is evaluated, the algorithm non-deterministically adds the condition or its negation to the corresponding path condition and checks the path condition's satisfiability using an "off-the-shelf" decision procedure. If the path condition becomes infeasible, the current execution terminates (i.e., the algorithm backtracks).

The approach applies both to (executable) models and to code. It generates an optimized test suite for flexible/user specified criteria, while it checks for errors during test generation. Moreover, it uses method pre-conditions (if available) to generate only test inputs that satisfy these preconditions.

Our symbolic execution framework is built on top of the JPF model checker. The model checker explores the symbolic execution tree of the analyzed program, using its usual state space exploration techniques. Essentially, a state includes a heap configuration, a path condition on primitive fields, and thread scheduling. Whenever a path condition is updated, it is checked for satisfiability using an appropriate decision procedure, such as the Omega library (Omega website) for linear integer constraints. If the path condition is unsatisfiable, the model checker backtracks. The framework can be used for test input generation and for finding counterexamples to safety properties. Model checking lends itself to test input generation, since one simply writes as a set of (temporal) properties that say that some coverage cannot be achieved and the model checker will find counterexamples and associated path conditions, if they exist, that encode program paths that achieve the stated coverage goal. The path conditions are solved to obtain the actual test inputs.

8.3.3 Tool Support and Applications

The approach described above has been implemented as an extension of the Java PathFinder model checking framework JPF-SE (Anand, Pasareanu, and Visser 2007) and is available from the JPF distribution site (JPF website). It can be used for test case generation for Java programs or for UML statecharts (which we model in Java). The original implementation required a source-to-source transformation, while our current implementation is implemented directly in JPF, by a non-standard execution of Java bytecodes. The symbolic information is stored in attributes that are associated with the program variables and the operands in the operand stack. JPF-SE's current implementation supports integer linear operations; implementation for reals (i.e., linear and non-linear constraints for real arithmetic) is ongoing work.

We have applied our framework for testing object-oriented code: in (Visser, Pasareanu, and Khurshid 2004; Visser, Pasareanu, and Pelanek 2006) we show how a combination of model checking, symbolic execution and abstraction can be used to generate test cases for a variety of Java container classes (including `TreeMap` and `BinomialHeap`). Test cases are sequences of method calls in the container interfaces that add and remove elements from those containers. In (Visser, Pasareanu, and Khurshid 2004; Visser, Pasareanu, and Pelanek 2006) we also provide experimental comparisons of these advanced techniques with random test case generation. All the data from the experiments are available with the JPF distribution (JPF website).

We have also applied our techniques to the Onboard Abort Executive (OAE), a prototype for Crew Exploration Vehicle (CEV) ascent abort handling being developed by NASA Johnson Space Center. The executive monitors a set of flight rules which, if violated, result in commanding specific kinds of aborts. The code for the OAE is not very large (only ~400 LOC) but it has complex conditions encoding different flight rule and abort combinations. Manual testing for the OAE is time consuming (e.g., it took several days for the developers of the OAE to generate tests that the developers believed achieved adequate coverage), while guided random testing did not cover all aborts. In contrast, it took two hours to set up JPF-SE for testing the OAE, and then in less than a minute it generated ~200 tests to cover all aborts and flight rules. Moreover, the generated test suite was used for regression testing, and it found a significant error in a new version of OAE that lead to redesign of the code.

8.4 Related Work

The work related to the topic of automated test case generation, symbolic execution, and model checking is vast, and for brevity we only highlight here some of the closely related work. We first summarize tools and technique) that generate test sequences for object-oriented programs. JTest (Parasoft website) is a commercial tool that generates test sequences for Java classes using "dynamic" symbolic execution, which combines concrete and symbolic execution over randomly generated paths. Although this approach may generate redundant tests and may not guarantee the desired degree of coverage, random testing is in many cases pretty effective (Visser, Pasareanu, and Pelanek 2006). The AsmLT model-based testing tool (Grieskamp *et al.* 2002) uses concrete state-space exploration techniques and abstraction mappings. Rostra (Xie, Marinov, and Notkin 2004) also generates unit tests for Java classes, using bounded-exhaustive exploration of sequences with concrete arguments and abstraction mappings.

The Korat tool (Boyapati, Khurshid, and Marinov 2002) supports non-isomorphic generation of complex input structures and requires the availability of constraints representing these inputs. Korat uses constraints given as Java predicates (e.g., repOK methods encoding class invariants). Similarly, TestEra (Marinov and Khurshid 2001) uses constraints given in Alloy to generate complex structures. The ASTOOT tool (Doong and Frankl 1994) requires algebraic specifications to generate tests (including oracles) for object-oriented programs. The tool generates sequences of interface events and checks whether the resulting objects are observationally equivalent (according to the specification).

Symstra (Xie *et al.* 2005) is a test generation tool that uses symbolic execution and state matching to generate test sequences for Java code. DART (Godefroid, Klarlund, and Sen 2005) and the related tools CUTE and JCUTE (Sen and Agha 2006) use an interesting combination of concrete and symbolic execution for test generation. A program is started on a randomly generated input, and symbolic constraints are collected along this concrete execution. These constraints are then negated (more precisely, one of the conjuncts in the collected constraints) and solved to obtain new inputs that are guaranteed to execute the program on alternative paths. A set of complementary approaches use optimization-based techniques (e.g., genetic algorithms) for automated test case generation (Tonella 2004; Baresel *et al.* 2004).

The work presented here is related to the use of model checking for test input generation (Ammann, Black, and Majurski 1998; Gargantini and Heitmeyer 1999; Heimdahl *et al.* 2003; Hong *et al.* 2002). The focus of these works is on specification-based test input generation (i.e., black-box testing) where coverage of the specification is the goal. Recently two popular software model checkers, BLAST (see section 7.4) and SLAM (see section 7.9), have been used for generating test inputs with the goal of covering a specific predicate or a combination of predicates (Beyer *et al.* 2004; Ball 2004). Both these tools use over-approximation based predicate abstraction and use some form of symbolic evaluation for the analysis of (spurious) abstract counterexamples and refinement. The work in (Ball 2004) describes predicate coverage as a new testing metric and ways to measure when the optimal predicate coverage has been achieved.

9 Case Studies

This section briefly overviews a few NASA-relevant case studies that employed program model checking.

9.1 DS-1 Remote Agent

(Havelund, Lowry, and Penix 2001) describes an application of SPIN to find errors in the AI multithreaded plan execution module aboard the Deep Space 1 (DS-1) spacecraft. DS-1 was a New Millennium spacecraft with an AI-based control system architecture called the Remote Agent. The plan execution module was based on an extension to Common Lisp called Executive Support Language (ESL). This layer provides reactive control mechanisms: It deals with managing tasks that require properties to hold at the start of and during task execution. If a required property becomes false during execution of a task, the task must be notified. This is clearly a multithreaded application with ample room for synchronization errors. Model construction was employed to model the ESL services in Promela, the language of the SPIN model checker. Lists were modeled as channels, and tasks as Promela processes. The environment was modeled as a parallel process that can change the status of a property. Two correctness properties were checked against the Promela model, a "release" property expressed as an assertion, and an "abort" property modeled in LTL. SPIN found four violations of these two properties in the Promela model; these violations were determined to reflect real errors (such as missing critical sections) in the implementation of ESL.

Though the errors found were corrected, other parts of the Remote Agent that were not analyzed contained a similar error, which caused the remote agent to deadlock in actual flight. The use of JPF and abstraction to find, after the fact, the defect that caused the in-flight error is described in (Havelund *et al.* 2000; Visser *et al.* 2003). A challenge experiment was performed to see whether model checking could find the error that led to a deadlock in space. The experiment involved a "front-end" group and a "backend" group, each ignorant of the defect, and was designed to mimic the process of an engineer doing visual inspection of code to find a suspect portion which needed further analysis. The front-end group was given access to the code of the Remote Agent executive. They isolated 700 lines of suspect code and gave it to the back-end group to model check, without telling the back-end group what they were suspicious of. The back-end group developed an abstract Java model and used the first-generation JPF (Havelund and Pressburger 2000) to model check the code. Their Java model changed an event counter to be modulo 3, to reduce the size of the state space. This was an "informal" abstraction because it was not shown to preserve errors. In another version, predicate abstraction was applied.

JPF revealed a deadlock which was in fact the error that led to the deadlock in space. The error was of the same kind as had been found using SPIN in 1997 on the other portion of the remote agent executive: a missing critical section. The buggy code involved a process containing a check for whether an event counter had registered any events; if it had not, it then waited for future events. However, this code was not in a critical section, so there was the unlikely possibility that after having checked the event counter, but before waiting, a new event could occur. The process would wait anyway, and this event would be lost. Another process was waiting on the first process to signal an event, so both ended up waiting for each other, causing the deadlock. You can see the relevant part of the Java model in the "oldclassic" example in the JPF distribution.

9.2 Rover Executive

This study (Brat *et al.* 2004) was a controlled experiment where several teams applied different V&V technologies to find seeded errors in an experimental Martian rover controller; model checking using JPF was one of the technologies. One of the results was that the original abstraction of time was an over-approximation so that errors found were spurious, but a deeper understanding of the controller code than possible under the time constraints was necessary to determine they were spurious. The JPF team decided to build an under-approximation so that no spurious errors would occur, so understanding the actual code was not necessary.

9.3 DEOS

DEOS (Dynamic Enforcement Operating System) is a realtime OS used in Honeywell's avionics product line. It enforces space and time partitioning; the latter means that "a thread's access to its CPU time budget cannot be impaired by the actions of any other thread." During formal inspection, a subtle bug was found by Honeywell's developers that broke time partitioning. An experiment was conducted to see whether a NASA team unfamiliar with the DEOS code and the error could rediscover the error using systematic model-checking techniques. They did.

About 1500 lines of the C++ code were analyzed, mapped nearly one-to-one to Promela. Predicate abstraction was also used. The most time-consuming operation was building a faithful environment modeling the threads, clock, and system timer. An ad-hoc iteration process was used to refine the environment based on analysis of spurious counterexamples. An environment was also systematically generated using filter-based techniques, with much less effort (Penix *et al.* 2005).

9.4 SAFM Case Study: NASA C++ flight software

This section provides:

- History of the Propel toolset used for model-checking C++ applications,
- A brief overview of a NASA C++ flight application selected for a case study,
- Rationale for the selection of this application, and

- A discussion of the case study.

Details of the case study relevant to model checking are mentioned in several sections of this guidebook. This case study is described in (Markosian, *et al.* 2007).

9.4.1 History of the Propel Toolset

Development of the Propel toolset for model checking C++ applications was started at NASA under funding from the Engineering for Complex Systems (ECS) program in 2001. The principal goal of this work was to develop tools for model checking NASA C++ flight software. The strategy for developing the tools was to leverage the existing research on program model checking, which had produced a research prototype tool, Java PathFinder (JPF), applicable to Java. The strategy of leveraging the prototype Java model checker was to be done by productizing the prototype and developing a translator from C++ to Java. JPF would then be applied to the translated C++ application.

It was recognized from the outset that not all of the C++ language and libraries could be mechanically translated into Java, and that there would likely be issues even with successfully translated code (for example, an increase in state space introduced by the translator). Nevertheless, this was deemed preferable to developing a completely new program model checker for C++. This work received continued funding from ECS through late 2004, at which time funding was terminated by a re-ordering of research priorities. By the time funding ended, a translation approach for the identified target subset of C++ had been designed and a prototype translator constructed. However, the prototype was not robust and no further funding was available to support its continued development.

9.4.2 Brief Overview of SAFM

The Shuttle Abort Flight Management system (SAFM) was developed by NASA Johnson Space Center and General Dynamics Decision Systems as part of the Shuttle Cockpit Avionics Upgrade (CAU). SAFM is a single-threaded application written in 30 KLOC of C++ that follows coding standards appropriate to safety-critical applications.

SAFM evaluates the potential abort options for the Space Shuttle under various contingencies and provides abort recommendations to the crew. Primarily as a result of the loss of Columbia and the subsequent re-focusing of NASA's crewed space flight away from the space shuttle and similar vehicles, both CAU and SAFM were canceled before being deployed, but the developers of SAFM remained interested in the Software Assurance Research Program (SARP) initiative's analyses of the software.

9.4.3 Rationale for Selection of SAFM as the Target Application

Under the ECS program, SAFM had been identified as one of several candidates for a feasibility study. Starting in 2004, under funding from NASA IVV Facility Software Assurance Research Program (SARP), we undertook a technology transfer activity to apply the Propel tools to a significant NASA C++ flight software application and to develop this Guidebook based on that and other case studies.

One of the first tasks under the SARP project was to investigate the previously identified candidate C++ applications, including SAFM, and select one. Since C++ had only recently been introduced for flight software applications at NASA (earlier flight software was generally written in Ada, assembler and C) our selections were limited. We chose SAFM for several reasons:

- High criticality—it is capable of determining a safe landing trajectory for the shuttle autonomously in case of a failure requiring an abort during ascent;
- Commonality of interest with the developers and their willingness and availability to work with us;
- Good conformance of the application with technical criteria for C++ model checking using Propel; and
- Manageable size (approx. 30,000 LOC).

High criticality—Advanced defect detection technologies such as model checking are thought to be capable of detecting subtle defects which can escape detection using standard testing approaches, but they also represent a departure from current standard practice and are relatively immature. Further NASA development and deployment of leading-edge V&V technologies is best justified by demonstrating their success on safety-critical NASA applications. SAFM was rated "Crit 1R"—high criticality but redundant software. The in-flight SAFM system is redundant because there are ground systems that duplicate and take priority over in-flight SAFM functionality in the presence of effective ground communication. In the absence of ground communication, the on-board SAFM system provides the required situational awareness to the crew. Since human life might depend on flawless operation of SAFM, it needs to be "human rated," which requires extensive V&V.

Commonality of interest—Success in applying a new software engineering technology in the context of a significant application requires a good working relationship among the application developers and the technology developers. The SAFM development team was willing to work with us for several reasons. They recognized a need for significantly greater autonomy (functionality achieved through automation in the absence of ground control) in space flight software than is currently available in applications like SAFM, and they recognized that a major barrier to greater autonomy has been the inability to perform verification and validation of autonomous software. Program model checking has been thought to be particularly well suited to V&V of complex applications such as autonomy software. In addition, although SAFM had been delivered to NASA by its developers, it was not scheduled for deployment until 2006; this two-year lead time would allow remediation of potential issues that our technologies might reveal.

Technical match—Mismatch between technology capabilities and a target application can result in redirection of project focus, resources lost in developing workarounds, or even a complete failure. SAFM was a good fit to the technical characteristics required by the translation approach taken by Propel. Most of the coding standards identified for successful model checking using Propel were met in SAFM. For example, SAFM did not make extensive use of the C++ Standard Template Library, and had only one use of multiple inheritance. On the other hand, SAFM lacked an important characteristic which would have made it a much better candidate for an early feasibility demonstration: It is single-threaded, whereas the properties that model checkers are best adapted to verifying “out of the box” include freedom from deadlock and other concurrency defects. Another characteristic of SAFM that impeded the effectiveness of model checking is that much of SAFM performs “number crunching” using floating point arithmetic. Model checking, which focuses on the correctness of state transitions, is not particularly effective at detecting defects in arithmetic computations.

Manageable size—Program model checking is limited by the size of the state space of complex applications. SAFM’s size was known from the start to be too large to allow program model checking in its entirety using the approach taken with the Propel toolset. This did not represent a critical problem because the application was modular and would allow model checking of selected modules. Our goal was not to model check, in production mode, a complete application, but to demonstrate the feasibility of Propel. Still, the size of the application needed to be limited because the entire application and its test environment would need to be ported, compiled, and executed on the platforms used by Propel developers at Ames Research Center; and the Propel team would need to obtain a sufficient understanding not only of the application, but also of its low-level technical requirements (the system requirements specification—SRS) to apply Propel effectively.

9.4.4 Property Identification

In order to perform model checking, you need to specify a number of properties that the application under analysis must satisfy. JPF detects certain generic properties by default—for example, “no assertion violations” and certain concurrency-related properties such as “no deadlocks” and “no race conditions.” The other type of properties that can be checked by JPF are application-specific properties. Since SAFM was a single-threaded application which by default did not contain any assertions, the only properties that we could concentrate on were application-specific properties that required domain knowledge. We worked closely with the SAFM developers, particularly the SAFM test personnel, and identified a number of those properties. The properties were then tested using the existing SAFM test suite. In some cases, this revealed violations of the initial properties so that iteration with the developers was necessary to correctly formulate the properties. In the more interesting cases, it revealed implementation defects in SAFM.

9.4.5 Code Instrumentation

Property oracles were inserted into the code using Aspect C++. An important side benefit of using Aspect C++ was that the application could be instrumented without modification of the source files. This allowed the case study to keep pace with successive SAFM releases.

9.4.6 Model Checking SAFM

Our intention was to use JPF for program model checking of these properties. The SAFM code was partially translated to Java using Propel, a C++-to-Java translator that was being developed for the purpose of bringing model checking and other program analyses to C++. Application of the model checker was limited for the reasons discussed above. However, an experiment was performed in model checking a portion of SAFM that contained an error found by inspection, but that had not been revealed by the SAFM test suite. A D4V change, that of exploiting polymorphism (see section 6.3.2), made the error more apparent. A property revealing the error was formulated and model checked on the original translated code and the code after performing the D4V change. JPF's choice generators were used to completely cover an input space relevant to the portion. On average, JPF used 25% less time and 13% less memory to check the code that was modified based on the identified D4V opportunity, even though the space explored was the same, because the paths to be checked to detect the error were shorter.

9.5 MER Arbiter

JPL's Mars Exploration Rover (MER) software is multithreaded software that must deal with shared resources. The arbiter module prevents potential conflicts between resource requests, and enforces priorities. It was the subject of a case study using SPIN described in (Holzmann and Joshi 2004) and has been mentioned in section 4.4 on Model-Driven verification. Model checking the original problem, with 11 threads and 15 resources, was deemed infeasible. In the study, three versions of a reduced problem with fewer threads and resources were modeled: a hand-built Promela model; a version using the original C code, with all relevant state information tracked using `c_track` (see section 4.4); and finally a version like the second but using data abstraction as described in section 4.4 with only essential data retained. SPIN found safety and liveness property violations in each version, but nothing considered serious by the engineers; thus the application of SPIN gave more confidence that the design was correct. The first model was efficient, but relies on the accuracy of the hand-built model. The second was inefficient, but adhered to the original C code. The third was efficient, yet still adhered to the original C code.

9.6 CEV Onboard Abort Executive Prototype

See section 8.3.3.

Appendix A – An Example of a Primitive Type Abstraction in C++

This appendix provides an example of a C++ class definition for values of type double. It supports the verification & validation of a number of important properties associated with different operations involving double value. It generates an assertion violation:

- If the number is not a legal number
- If the number represents a +infinity or a –infinity
- If an attempt is made to divide a double number by zero

It encapsulates the definition of various operations performed on double numbers and includes assertions that are necessary to check for the above violations. A developer or a V&V person trying to apply a tool such as a model checker can use this class to quickly check if the target application violates any of the above properties.

Note that this class can be extended to include other checks; for example, operations involving exact comparison between double values. One would need to extend the class to include the definition of `operator==()` and a constructor that also takes an epsilon value as a parameter. This is left as an exercise for the reader.

```
#ifndef VTYPES
#define VTYPES

#ifdef __cplusplus

#include <stdio.h>
#include <math.h>
#include <assert.h>

class Double {
   double m_value;

public:

   Double () {
      m_value = NAN;
   }

   Double (double value) : m_value(value){
   }

   Double& operator= (double value) {
      m_value = value;
      return *this;
   }

   Double operator+ (double value) {
      assert(!isnan(value));
      assert(!isnan(m_value));
      assert(!isinf(value));
      assert(!isinf(m_value));

      return Double(m_value + value);
   }

   Double operator/ (double value) {
      assert(!isnan(value));
      assert(!isnan(m_value));
      assert(!isinf(value));
      assert(!isinf(m_value));
      assert(value != 0.0);

      return Double(m_value / value);
  }

   Double operator- (double value) {
      assert(!isnan(value));
      assert(!isnan(m_value));
      assert(!isinf(value));
      assert(!isinf(m_value));

    return Double(m_value - value);
  }

   Double operator* (double value) {
      assert(!isnan(value));
      assert(!isnan(m_value));
      assert(!isinf(value));
      assert(!isinf(m_value));

      return Double(m_value * value);
   }

   operator const double () {
      return m_value;
   }
};

#else  // not __cplusplus
```

```
typedef double Double;

#endif

#endif VTYPES
```

Appendix B – Example Implementation of Minimal C++ PbC Infrastructure

```
/* @file test.cpp - example of how to use the simple
 *                  programing-by-contract system for C++
 */

#include "pbc"

class Foo {
   double r;

public:
   Foo () : r(40.0) {}

   double doThat (double d);

   // if we have class invariants, those have to be implemented in
   // a dedicated 'invariant' member function
   bool invariant () const {
      return ((r >= 0.0) && (r <= 42.0));
   }
};

double Foo::doThat (double d) {

   MBR_CONTRACT_RP( double, Foo, doThat, // type, class, function
                    double, d, // checked arg types and names
                    REQUIRES( d > 0), // precond
                    ENSURES((d==old_d) && (result<40.0)) // postcond
                  );

   if (d < 1.0) {
      r += d * 2.0;
      RETURN(0.0);  // use the RETURN() macro instead of 'return'
   } else {
      r += d;
   RETURN(r);
   }
}
```

```
class Bar {
public:
   void doThis(int cmd);

   // if we don't have invariants, but have functions using
   // contracts, we have to define an invariant placeholder
   NO_INVARIANT
};

void Bar::doThis (int cmd) {

   MBR_CONTRACT_P( Bar, doThis,                // no return type
                   int, cmd,                   // checked argument
                   REQUIRES( cmd > 0),
                   NO_ENSURES                  // no post-conditions
                  );

   Foo f;
   f.doThat(5.0);
}

//// test driver
int main (int argc, char* argv[]) {
   Bar b;
   b.doThis(42);

   return 0;
}

/* @file pbc - header file for simple programming-by-contract
 * support for C++
 *
 *    THIS CODE IS ONLY MEANT TO BE USED AS AN EXAMPLE
 *
 * The principle is to use macros defining function nested contract
 * classes that check for preconditions in the constructor and
 * postconditions in the destructor, using automatic contract
 * objects / RAII to make sure all exit points of a function are
 * instrumented. Pre- and post conditions are defined by means of
 * macros expanding into member functions of the contract class.
 * 'Old' value copies and references to input parameters are
 * kept as data members of the contract class. Invariants have to be
 * defined as const member functions of the owning class. The base
 * class for all contracts mainly keeps track of all contract
 * objects on the stack, to support debugging contract violations
 *
 * This version does not support sub-contracting and interface (pure
 * virtual function member) contracts. It also does not support
 * multi-threading, but can be extended to do so.
 *
 */

#ifndef _PBC
#define _PBC

#include <iostream>

using namespace std;

//// the base class for all contracts
class contract_t {

protected:
   contract_t* caller;      // contract object of the caller
   const char* class_name;  // class name of contract owner
   const char* func_name;   // the function member owning the
                            // contract
   bool in_postcond;

  static contract_t* last; // last active contract object created

  contract_t (const char* class_name, const char* func_name) :
             class_name(class_name),
             func_name(func_name),
             in_postcond(false) {
      // for multi-threading, we would have to be a bit more
      // sophisticated (keeping a map of stack adr bases)
      caller = last;
      last = this;
   }

   virtual ~contract_t () {
      last = caller;
   }

public:
   const char* get_class_name() {  return class_name; }
   const char* get_func_name() { return func_name; }
   contract_t* get_caller() { return caller; }
   virtual void report (ostream& serr) = 0;
};


//// pointer to handler function for contract violations
```

```
extern void (*contract_handler) (contract_t* c,
                                 const char* file, int line,
                                 const char* contract_type,
                                 const char* cond);


//// do we want PbC instrumentation
#ifdef PBC

//// helper (embedded) macros for concrete contract member definition

#define REQUIRES(cond) \
   void precond () { \
      if (!(owner->invariant())) contract_handler(this, \
                                    __FILE__, __LINE__, \
                          "invariant", "precondition"); \
      if (!(cond)) contract_handler(this, __FILE__, __LINE__, \
                                      "precondition", #cond); \
   }

#define ENSURES(cond) \
   void postcond () { \
      if (!(cond)) contract_handler(this, \
                                  __FILE__, __LINE__, \
                             "postcondition", #cond); \
      if (!(owner->invariant())) contract_handler(this, \
                                  __FILE__, __LINE__, \
                         nvariant", "postcondition"); \
   }

#define NO_ENSURES ENSURES(true)

#define NO_REQUIRES REQUIRES(true)

#define NO_INVARIANT \
   bool invariant() const { return true; }

//// to be used instead of 'return'
#define RETURN(result) \
   return _contract_.set_result(result)


//// macros to define concrete contract classes inside of functions
//// and function members

//// contract for member func with non-void type and one
//// checked parameter
#define MBR_CONTRACT_RP( return_t, cname, fname, \
                       arg0_t, arg0, _pre, _post)\
   class _contract_t : public contract_t { \
   public: \
      const cname *owner; \
      return_t result; \
      const arg0_t  &arg0, old_##arg0; \
\
      _contract_t (const char* owner_class, const cname* owner, \
                 const char* func_name, arg0_t &arg0) \
               : contract_t(owner_class, func_name), \
                 owner(owner), arg0(arg0), old_##arg0(arg0) { \
         precond(); \
      } \
\
   return_t set_result (return_t r) { \
      return (result = r); \
   } \
\
   ~_contract_t () { \
      in_postcond = true; \
      postcond(); \
   } \
\
   void report (ostream& serr) { \
      serr << class_name << "::" << func_name \
           << "(" << old_##arg0 << ")"; \
      if (in_postcond) { \
         serr << " : (" << arg0 << ") -> " << result; \
      } \
      serr << endl; \
   } \
\
   _pre \
   _post \
  } _contract_(#cname, this, #fname, arg0);


//// contract for void member func with one checked parameter
#define MBR_CONTRACT_P( cname, fname, arg0_t, arg0, _pre, _post) \
   class _contract_t : public contract_t { \
   public: \
      const cname *owner; \
      const arg0_t  &arg0, old_##arg0; \
\
      _contract_t (const char* owner_class, const cname* owner, \
                 const char* func_name, arg0_t &arg0) \
               : contract_t(owner_class, func_name), \
                 owner(owner), arg0(arg0), old_##arg0(arg0) { \
      precond(); \
   } \
\
   ~_contract_t () { \
     postcond(); \
   } \
\
   void report (ostream& serr) { \
      serr << class_name << "::" << func_name \
           << "(" << old_##arg0 << ")" << endl; \
   } \
\
      _pre \
      _post \
   } _contract_(#cname, this, #fname, arg0);

//// ... more contract class definition macros


#else // no PbC instrumentation

#define REQUIRES(cond)
#define ENSURES(cond)
#define NO_INVARIANT
#define NO_ENSURES
#define NO_REQUIRES
#define RETURN(result) return result
#define MBR_CONTRACT_RP(return_t,cname,fname,arg0_t,arg0,_pre,_post)
#define MBR_CONTRACT_P( cname, fname, arg0_t, arg0, _pre, _post)

#endif // PbC support

#endif _PBC
```

Appendix C – Acronyms and Abbreviations

AOP	aspect-oriented programming
BBFS	Bounded Breadth-First Search
BDFS	Bounded Depth-First Search
BEG	Bandera Environment Generator
BFS	Breadth-First Search
BLAST	Berkeley Lazy Abstraction Software Verification Tool
CAU	Cockpit Avionics Upgrade
CEGAR	Counter-Example Guided Abstraction Refinement
CICT	Communication, Information, and Computation Technology
CTL	Computation Tree Logic
DEOS	Dynamic Enforcement Operating System
DFS	Depth-First Search
DS-1	Deep Space 1
ECS	Engineering for Complex Systems
ESL	Executive Support Language
ETDP	Exploration Technology Development Program
FSM	Finite State Machine
IDE	integrated development environment
IMA	Integrated Modular Avionics
ITSR	Information Technology Strategic Research
IV&V	Independent Verification and Validation
JML	Java Modeling Language
JNI	Java Native Interface
JPF	Java PathFinder
JVM	Java Virtual Machine
KLOC	thousand lines of code

LTL	Linear Temporal Logic
MC/DC	modified condition/decision coverage
MER	Mars Exploration Rover
MJI	Model Java Interface
MPL	Mars Polar Lander
OAT	Office of Aerospace Technology
OOP	object-oriented programming
OSMA	Office of Safety and Mission Assurance
PbC	Programming by Contract
Promela	Process Meta Language
QRE	Quantified Regular Expressions
RSE	Robust Software Engineering
SAFM	Shuttle Abort Flight Management system
SARP	Software Assurance Research Program
SDV	Static Driver verifier
SPIN	Simple Promela Interpreter
SRS	system requirements specification
SUT	system under test
UML	Unified Modeling Language
V&V	Verification and Validation
VM	virtual machine

Appendix D – Glossary

advice. In aspect-oriented programming, a function, method, or procedure applied to a program at a join point.

all-DU-paths. DU = “definition-use”. A *DU path* is a data flow path from a point where a variable is assigned a value (its *definition*) and a point where the value assigned is used. *All-DU-paths* is a test coverage metric.

application-specific property. A program property that depends on the nature of the application and typically contains domain-specific information. For example, in a particular program, following the operation x := y / z , “ x < 985000” may be an application-specific property, whereas at another point or in another program, following the operation x := y / z , “ x > 985000” may be the correct property.

approximation. A model that is an abstraction of a concrete system. Approximations are used where the actual (concrete) state space is too large or complex to model check.

aspect. A cross-cutting concern; that is, a concern that may cut across many modules. Cross-cutting concerns are often contrasted with core concerns, which are the main concerns of the program and are modularized by classes. Logging is an example of an aspect; logging may cross-cut many modules.

aspect-oriented programming (AOP). Languages that allow the programmer to express, in stand-alone modules called aspects, features or behaviors that cut across many modules in a program.

assertion; assert statement. A statement that asserts that a program property is expected to be true of the state at that point in the program. This can sometimes be verified by a model checker.

assume guarantee paradigm. A divide-and-conquer mechanism for decomposing the task of verifying property *P* about a system into subtasks about the individual components of the system. Components are checked not in isolation, but in conjunction with assumptions (*A*) about the context of the components. Property *P* is said to be the “guarantee” that a component provides under “assumption” *A*.

atomic section. A set of statements that is tracked as a single transition in model checking. In Java PathFinder, an atomic section is a sequence of instructions that is executed by the model checker without a thread context switch.

Bandera Environment Generator (BEG). A tool set that integrates existing programming language processing techniques with newly developed techniques to provide automated support for the extraction of safe, compact, finite-state environment models from Java source code. These models are then used during the verification of software components.

Berkeley Lazy Abstraction Software Verification Tool (BLAST). A software model checker for C programs. It uses a particular abstraction called “predicate abstraction” and abstraction refinement based on analysis of abstract counterexamples. The goal of BLAST is to check that software satisfies behavioral requirements that are encoded as safety properties.

best-first search. A variation of breadth-first search that explores the most promising neighboring state chosen according to an evaluation function.

bit state hashing. A lossy state compression technique that uses memory as a bit vector to encode a state. The encoding is compared with the encodings for previously visited states to determine whether a state has been visited previously. Also known as *supertrace*.

bounded breadth-first search (BBFS). A breadth-first search that imposes a maximum limit on the number of neighboring states searched.

bounded depth-first search (BDFS). A depth-first search that imposes a maximum limit on the depth of the search.

branch point. Choices resulting from conditional statements in the code, scheduling decisions, or different input values, which can produce multiple possibilities for the next state to be considered by the model checker.

branching factor. The number of neighboring states in a search.

breadth-first search (BFS). A search strategy where the model checker begins at the start state and explores all its outgoing transitions to the neighboring states, then for each of those nearest states, it explores all of its outgoing transitions to its neighbor states, and so on, until it hits a goal state or a state with no outgoing or unexplored transition.

bytecode. A binary representation of an executable program designed to be executed by a virtual machine rather than by dedicated hardware.

checkpoint. A point in the code where the program is in a consistent state and a property oracle can be inserted.

class invariant. An assertion, used in programming-by-contract, that is evaluated before and after each invocation of a public function. The expectation is that the assertion is true at both these execution points.

computation tree logic (CTL). A form of logic whose model of time is a tree-like structure in which there are multiple possible future paths; the path to be realized in the future is not determined.

control abstraction. The removal of program components that are irrelevant to the property under analysis.

concurrent system. A system in which multiple threads are executing at the same time and can interact with each other.

concurrency error. An error caused by conflicting behavior of multiple processes in a concurrent system.

counterexample. In the context of model checking, a response from a model checker that shows the violation of a property and the major events leading to that violation. Also called an *error trace*.

Counter-Example Guided Abstraction Refinement (CEGAR). An approach to automatic abstraction and refinement of software systems that analyzes abstract counterexamples to derive progressively better models of the program being verified.

data abstraction. In model checking, the replacement of the large domain of a program variable with a smaller domain; for example, using the *signs* abstraction to reduce the very large domain of an integer variable to three values { negative, zero, positive }.

deadlock. In a concurrent system, a state in which two or more processes wait for each other to finish, and hence neither can proceed. There is no exit transition from a deadlock state.

depth-first search (DFS). A search strategy in which the model checker begins at the start state and recursively explores one of its outgoing transitions to a successor state (i.e., as far as possible along each branch) before backtracking and selecting the next transition.

directed model checking. The use of guided or heuristic search techniques during state space exploration.

driver. A program component that invokes operations on the unit under test.

d-step. A language statement used in a Promela model to execute a number of statements in a single step. If one of the statements blocks, it is considered a runtime error.

Dynamic Enforcement Operating System (DEOS). A real-time operating system developed by Honeywell Laboratories for integrated modular avionics systems.

edge. A transition between states (nodes) in a state machine.

environment assumption. A filter that captures knowledge about the usage of software components to help refine a naively generated test environment. These assumptions must be checked against implementations of the missing components (that represent the actual environment) to ensure that the entire system satisfies the desired properties.

environment model. An abstract runtime context in which the component under test executes, containing both data information and control information that influence the unit under analysis.

error trace. A response from the model checker that shows the violation of a property and the major events leading to that violation. Also called a *counterexample*.

Executive Support Language (ESL). A plan execution language developed at the Jet Propulsion Laboratory to support the construction of reactive control mechanisms for autonomous robots and spacecraft.

finite state machine (FSM). A model of system behavior composed of a finite number of states and transitions between those states.

generic property. A property that is not dependent on the nature of the application being checked. Generic properties can be formulated and understood by those who do not have any application-specific domain knowledge. An example of a generic property is "the system has no deadlock states."

heuristic search. A search process that exploits the information known about the system under analysis in order to accelerate the search process.

Java Modeling Language (JML). An open source language extension for Java that specifies behavioral interfaces of modules. It provides additional expressions focused on avoiding side effects that might occur if the assertions were simply written in Java.

Java Native Interface (JNI). A programming interface that allows Java code running in the Java virtual machine to interact with native applications and libraries written in other programming languages.

Java PathFinder (JPF). An open source model checker developed at NASA Ames Research Center as a verification and testing environment for Java programs.

join point. In aspect-oriented programming, a point where additional behavior, in the form of *advice,* can be applied to change the behavior of the program.

inheritance. In object-oriented programming, a hierarchical specialization of user-definable data types (called "classes") in terms of data and function members.

instrumentation. Insertion of code in a program to monitor or measure the performance of the program, typically for detecting, diagnosing and logging error information. For program model checking, instrumentation typically takes the form of annotations and code extensions that are inserted in the source. Aspect-oriented programming provides alternative methods of instrumenting a program using advice and join points.

integrated development environment (IDE). A software application that provides comprehensive facilities to computer programmers for software development. An IDE typically includes a source code editor and tools for compiling and debugging the program.

iterative widening. Repeated searches with increasing values for the width of the search.

linear temporal logic (LTL). A specification language used to express properties that hold for all paths through the state space. In LTL one can encode formulas about the future of paths such as that a condition will eventually be true, a condition will be true until another condition becomes true, etc.

liveness property. An assertion that something good (a desired action) happens or keeps happening. Liveness properties are used mainly to ensure computational progress; that is, that under certain conditions, some event will ultimately occur.

loop bound. An assertion that checks that a loop can be traversed only a bounded number of times.

loop invariant. An assertion that is true at the beginning of the loop and after each execution of the loop body.

loop variant. An assertion that describes how the data in the loop condition are changed by the loop. It is used to check forward progress in the execution of loops to avoid infinite loops and other incorrect loop behavior.

model. A abstract representation of the structure and behavior of a system.

model checking. A collection of techniques for analyzing a model of a system to determine the validity of one or more properties of interest.

model-driven verification. A technique that uses abstraction mappings during concrete model checking to analyze an under-approximation of the feasible behaviors of a system. All reported counterexamples correspond to real errors.

model extraction. Translation of the system being modeled into the input notation of an existing model checker.

modeling primitive. One of the fundamental building blocks of a model.

multiple inheritance. In object-oriented programming languages, inheritance from more than one parent class.

node. A state in a model.

object-oriented programming (OOP). A programming paradigm that separates features and behaviors into classes of objects and methods that act on these classes.

over-approximation. The result of abstracting a concrete system when there are more behaviors in the abstraction than in the concrete system. Over-approximation preserves "true" results for safety properties in the sense that, if a safety property is shown to hold in the abstract system, then it also holds in the concrete system. However, violations reported for the abstract system may be spurious; that is, they may not occur in the concrete system.

partial coverage. Checking a portion of the state space of a system.

pointcut. In aspect-oriented programming, a set of join points where additional code, called *advice*, is executed.

pointer arithmetic. The use of arithmetic operations to modify a pointer's target address.

polymorphism. The ability of objects of different types to react in a type specific way to method calls of the same name.

postcondition. An assertion that is evaluated with the intent of evaluating to true just after the execution of a function or method. In particular the predicate is evaluated even if the function or method includes exception control flow. Postconditions are used particularly in programming-by-contract.

precondition. An assertion evaluated before a function or is executed. Preconditions are used particularly in programming-by-contract.

predicate abstraction. A special case of an over-approximating data abstraction which maps a potentially infinite-state program into a finite-state program, via a finite set of predicates over the program's variables.

preprocessor directives. Lines of code executed by the preprocessor before the actual program is compiled. The preprocessor directives instruct the compiler how to treat the source code.

primitive typecasting. In C and C++, casting between primitive types.

program abstraction. Transformation of a program into another program that still has some key properties of the original program, but is much simpler, and therefore easier to analyze. In model checking, abstractions are used to reduce the size of a program's state space.

program model checking. Application of model-checking techniques to software systems, and in particular to the final implementation where the code itself is the target of the analysis.

program slicing. Reduction of the behaviors of a program by removing control points, variables, and data structures that are deemed irrelevant to checking a given property.

Programming by Contract (PbC). An extension of assertions providing specialized assertion types by defining standard scopes and evaluation rules for them, especially in the context of object-oriented programming. The main purpose of contracts is to clearly define responsibilities between the caller and the callee. The caller's responsibility is to fulfill preconditions and the callee's responsibility is to fulfill the postconditions. The goal of PbC is to help to avoid errors that are caused by ambiguity of responsibility.

Promela (*Pro*cess *Me*ta *La*nguage). A powerful C-like specification language with a variety of synchronization primitives that enables construction of models of distributed systems. Promela is the modeling language used by the SPIN model checker.

property. A precise condition that can be checked in a given state or across a number of states, usually expressed as a logical specification.

property oracle. A program or a piece of code that determines whether or not a property holds during model checking.

property preservation. A property is preserved under an abstraction of a program if the property evaluates to the same value when checked in both the abstraction and the program.

random search. A search strategy that non-deterministically selects a successor to explore.

reactive state machine. A state machine model of an application that reacts to environmental or internal stimuli.

safety property. An assertion that nothing bad will happen during the execution of the program. Safety properties are used mainly to ensure that under certain conditions, an event will never occur.

slicing criteria. Given a program and a set of statements of interest in the program, called the slicing criteria, a program slicer computes a reduced version of the program by removing program statements that do not affect the computation at the statements in the criteria.

SLAM. A Microsoft project that combines static analysis with model checking and deductive reasoning. The main goal of SLAM is to check temporal safety properties of C programs while minimizing false positives.

SPIN. A model checker developed by Gerard Holzmann and used for analyzing the logical consistency of concurrent and distributed systems. The tool's main focus is on proving the correctness of process interactions.

state (of a program). The heap, together with all thread stacks and thread execution states.

state property. A property that holds for an individual state in isolation from other states.

state space. The collection of all possible program states.

state space explosion. The exponential increase in the state space with every variable or thread added to the system.

state storage. A mechanism that allows a model checker to remember the previous states that it has visited

static analysis. The evaluation of source code for defects without actually executing the code.

stub. A program component that replaces operations invoked by the unit under test.

symbolic execution. A technique that uses symbolic values, instead of actual data, as input values and represents the values of program variables as symbolic expressions.

temporal property. A system property that relates state properties of distinct states in the state space to each other along possible execution paths. Also known as a *dynamic property*.

thread interleaving. In multi-threaded programs running on a single CPU, an interleaved execution order of the statements of the threads. Usually there may be more than one possible interleaving. The actual interleaving depends on the thread scheduler, and is not predetermined for each execution. Thread interleaving greatly increases the state space of a system.

transition. A state change in a program.

under-approximation. An abstraction of a concrete system in which there are fewer behaviors than in the concrete system. Under-approximation preserves "false" results for safety properties in the sense that, if a property is shown to be violated in the abstract system, then it is also violated in the concrete (un-abstracted) system. However, if the property holds in the abstract system, it does not necessarily hold in the concrete system.

union. In C and C++, a pseudo type used for memory that can be accessed with alternative concrete type definitions. Unions are not supported by languages like Java.

unit. The smallest testable part of an application.

unit testing. A procedure used to validate that individual units of source code are working properly.

untyped allocation. In C and C++, memory allocated by malloc/calloc. Untyped allocators do not consider types, only allocation size in bytes.

universal environment. An environment model capable of invoking (or refusing) any operation to or from the unit, in any order.

virtual machine (VM). An environment, usually a program or operating system, which does not physically exist but is created within another environment.

References

Abrahams, D., L. Crowl, T. Ottosen, and J. Widman. 2005. Proposal to add Contract Programming to C++.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1773.html

Allen, R.J., D. Garlan, and J. Ivers. 1998. Formal modeling and analysis of the HLA component integration standard. In *Proceedings of the 6th ACM/SIGSOFT Symposium on the Foundations of Software Engineering* (FSE 1998).

Ammann, P., P.E. Black, and W. Majurski. 1998. Using model checking to generate tests from specifications. In *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods*.

Anand, S., C. Pasareanu, and W. Visser. 2007. JPF-SE: A symbolic execution extension to Java PathFinder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS 2007).

Atlee, J. and J. Gannon. 1993. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1): 24–40.

Ball, T. 2004. A theory of predicate-complete test coverage and generation. Microsoft Research Technical Report MSR-TR-2004-28.

Ball, T., R. Majumdar, T. Millstein, and S.K. Rajamani. 2001. Automatic predicate abstraction of C programs. *SIGPLAN Notices*, 36(5): 203–213.

Ball, T., A. Podelski, and S.K. Rajamani. 2002. Relative completeness of abstraction refinement for software model checking. In *Tools and Algorithms for the Construction and Analysis of Systems: 8th International Conference* (TACAS 2002), pp. 158–172. Lecture Notes in Computer Science 2280. Berlin: Springer-Verlag.

Baresel, A., M. Harman, D. Binkley, and B. Korel. 2004. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2004), pp. 108–118.

BEG (Bandera Environment Generator) website. http://beg.projects.cis.ksu.edu/

Bezier, B. 1990. *Software Testing Techniques*, Second Edition. New York: Van Nostrand Reinhold.

Bensalem, S., Y. Lakhnech, and S. Owre. 1998. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of the International Conference on Computer-Aided Verification* (CAV 1998), pp. 319–331. Lecture Notes in Computer Science 1427. Berlin: Springer-Verlag.

Beyer, D., A.J. Chlipala, T. Henzinger, R. Jhala, and R. Majumdar. 2004. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering* (ICSE 2002).

BLAST website. http://mtc.epfl.ch/software-tools/blast/

BOOP website. http://boop.sourceforge.net/

Boyapati, C., S. Khurshid, and D. Marinov. 2002. Korat: Automated testing based on Java predicates. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2002), pp. 123–133.

Brat, G., D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, W. Visser, and R. Washington. 2004. Experimental evaluation of verification and validation tools on Martian rover software. *Formal Methods in Systems Design*, 25(2–3): 157–198.

Callahan, J., F. Schneider, and S. Easterbrook. 1996. Specification-based testing using model checking. In *Proceedings of the 1996 SPIN Workshop*.

Chaki, S., E.M. Clarke, A. Groce, S. Jha, and H. Veith. 2004. Modular verification of software components in C. Paper presented at the International Conference on Software Engineering (ICSE 2003). *IEEE Transactions on Software Engineering*, 30(6): 388–402.

Chan, W., R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese. 1998. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7): 498–519.

Chan, W., R.J. Anderson, P. Beame, D.H. Jones, D. Notkin, and W.E. Warner. 1999. Decoupling synchronization from local control for efficient symbolic model checking of statecharts. In *Proceedings of the 1999 International Conference on Software Engineering* (ICSE'99), pp. 142–151.

Chandra, S., P. Godefroid, and C. Palm. 2002. Software model checking in practice: an industrial case study. In *ICSE 2002: Proceedings of the International Conference on Software Engineering*, pp. 431–441.

Clarke, E.M., E.A. Emerson, and A.P. Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems* (TOPLAS), 8(2): 244–263.

Clarke, E.M., O. Grumberg, and D.E. Long. 1994. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems* (TOPLAS), 16(5): 1512–1542.

Clarke, E.M., O. Grumberg, and D.A. Peled. 2000. *Model Checking*. Cambridge, MA: MIT Press.

Clarke, L.A. 1976. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, 2(3): 215–222.

Corbett, J.C. 1998. Constructing compact models of concurrent Java programs. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1–10.

Corbett, J.C., M.B. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. 2000. Bandera: Extracting finite-state models from Java source code. In *Software Engineering 2000. Proceedings of the 2000 International Conference on Software Engineering*, pp. 439–448.

Corbett, J.C., M.B. Dwyer, and J. Hatcliff. 2000. Bandera: A source-level interface for model checking Java programs. In *Software Engineering 2000. Proceedings of the 2000 International Conference on Software Engineering*, pp. 762–765.

Cornett, S. 1996. Code Coverage Analysis. BullseyeCoverage website. http://www.bullseye.com/coverage.html

Cousot, P. and R. Cousot. 1997. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 238–252. New York: ACM Press.

Cousot, P. and R. Cousot. 1999. Refining model checking by abstract interpretation. *Automated Software Engineering Journal*, special issue on Automated Software Analysis 6(1): 69–95.

Coverity website. http://www.coverity.com/

Dams, D. 1996. *Abstract Interpretation and Partition Refinement for Model Checking*. Unpublished Ph.D. Thesis, Eindhoven University of Technology, The Netherlands.

Dang, Z, and R.A. Kemmerer. 1999. Using the ASTRAL model checker to analyze mobile IP. In *Proceedings of the 21st International Conference on Software Engineering* (ICSE 99), pp. 132–141. Piscataway, NJ: IEEE Press.

Digital Mars website (D language). http://www.digitalmars.com/d/

Doong, R.-K., and P.G. Frankl. 1994. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology* (TOSEM), 3(2): 101–130.

Dor, N., M. Rodeh, and S. Sagiv. 1998. Detecting memory errors via static pointer analysis (preliminary experience). In *Proceedings of SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 27–34.

dSPIN website. http://www-verimag.imag.fr/~iosif/dspin/

Drusinsky, D. 2000. The Temporal Rover and the ATG Rover. In *Proceedings of SPIN'00: SPIN Model Checking and Software Verification*, pp. 323–330. Lecture Notes in Computer Science 1885. Berlin: Springer-Verlag.

Dupuy, A. and N. Leveson. 2000. An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In *Proceedings of the 2000 Digital Aviation Systems Conference* (DASC), vol. 1, pp. 1B6/1–1B6/7.

Dwyer, M.B., G.S. Avrunin, and J.C. Corbett. 1988. Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pp. 7–15.

Dwyer, M.B., G.S. Avrunin, and J.C. Corbett. 1999. Patterns in property specifications for finite-state verification. In *Proceedings of 21st International Conference on Software Engineering*, pp. 411–420.

Dwyer, M.B., J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, H. Zheng, and W. Visser. 2000. Tool-supported program abstraction for finite-state verification. In *Proceedings of 23rd International Conference on Software Engineering*, pp. 177–187.

Edelkamp, S. 2006. Directed model checking: A tutorial. In *Proceedings of the 2006 SPIN Workshop*. http://andorfer.cs.uni-dortmund.de/~edelkamp/Directed Model Checking.ppt

Edelkamp, S., A. Lluch-Lafuente, and S. Leue. 2004. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology*, 5(2–3): 247–267.

Eiffel Software website. http://www.eiffel.com/

Evans D. 1996. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language design and Implementation* (PLDI'96), pp. 44–53.

FeaVer Feature Verification System website. http://cm.bell-labs.com/cm/cs/what/feaver/

FindBugs website. http://findbugs.sourceforge.net/

Gargantini, A. and C. Heitmeyer. 1999. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6): 146–162.

Godefroid, P. 2005. Software model checking: the VeriSoft approach. *Formal Methods in System Design*, 26(2), pp. 77–101.

Godefroid, P., N. Klarlund, and K. Sen. 2005. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2005), pp. 213–223.

Graf, S. and H. Saïdi. 1997. Construction of abstract state graphs with PVS. In Orna Grumberg (ed.), *Proceedings of Computer Aided Verification, 9th International Conference* (CAV'97), pp. 72–83. Lecture Notes in Computer Science 1254. Berlin: Springer-Verlag.

Graf, S. 1999. Characterization of a sequentially consistent memory and verification of a cache memory by abstraction. *Distributed Computing*, 12(2-3): 75–90.

GrammaTech website. http://www.grammatech.com/

W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. 2002. Generating finite state machines from abstract state machines. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2002), pp. 112–122.

Groce, A. and W. Visser. 2002. Model checking Java programs using structural heuristics. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 12–21.

Guerreiro, P. 2001. Simple support for design by contract in C++. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems* (TOOLS39), p. 24. Washington, DC: IEEE Computer Society.

Havelund, K. and N. Shankar. 1996. Experiments in theorem proving and model checking for protocol verification. In *FME'96: Industrial Benefit and Advances in Formal Methods*, pp. 662–681.

Havelund, K., and T. Pressburger. 2000. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4): 366–381.

Havelund, K., M. Lowry, and J. Penix. 2001. Formal analysis of a space-craft controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8): 749–765.

Havelund, K., M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. White. 2000. Formal Analysis of the Remote Agent Before and After Flight. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*.

Heimdahl, M.P.E., S. Rayadurgam, W. Visser, D. George, and J. Gao. 2003. Auto-generating test sequences using model checkers: A case study. In *Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software* (FATES).

Heitmeyer, C., J. Kirby, B. Labaw, and R. Bharadwaj. 1998. SCR*:A toolset for specifying and analyzing software requirements. In *Proceedings of the 10th International Computer-Aided Verification Conference*, pp. 526–531.

Henzinger, T.A., R. Jhala, R. Majumdar, and G. Sutre. 2002. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, pp. 58–70.

Holzmann, G.J. 1990. *Design and Validation of Computer Protocols.* Upper Saddle River, NJ: Prentice-Hall.

Holzmann, G.J. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering,* 23(5): 279–295.

Holzmann, G.J., and M.H. Smith. 2002. An automated verification method for distributed systems software based on model extraction. *IEEE Transactions on Software Engineering*, 28(4): 364–377.

Holzmann, G.J. 2004. *The SPIN Model Checker, Primer, and Reference Manual*. Boston, MA: Addison Wesley.

Holzmann, G.J., and R. Joshi. 2004. Model-driven software verification. In *Proceedings of the 11th SPIN Workshop*, pp. 77–92. Lecture Notes in Computer Science 2989. Berlin: Springer-Verlag.

Holzmann, G.J. 2006. The power of ten: rules for developing safety critical code. *IEEE Computer*, 39(6): 95-97.

Hong, H., I. Lee, O. Sokolsky, and H. Ural. 2002. A temporal logic based theory of test coverage and generation. In *Proceedings of the 8th International Conference on Tools and Algorithms for Construction and Analysis of Systems* (TACAS 2002).

Hwang, G., K. Tai, and T. Huang. 1995. Reachability testing: An approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering*, 5(4): 493–510.

Java Modeling Language (JML) website. http://www.cs.iastate.edu/~leavens/JML/

Java PathFinder website. http://javapathfinder.sourceforge.net/

JCAT website: http://www.dai-arc.polito.it/dai-arc/auto/tools/tool6.shtml

JUnit website: http://www.junit.org/

Jones, C.B. 1983. Specification and design of (parallel) programs. In *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pp. 321–332.

Kaner, C., J. Falk, and H.Q. Nguyen. 1993. *Testing Computer Software, Second Edition*. International Thomson Computer Press.

Kernighan, B. and D. Ritchie. 1998. *The C Programming Language*. Upper Saddle River, NJ: Prentice-Hall.

Khurshid, S, C. Pasareanu, and W. Visser. 2003. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

King, J.C. 1976. Symbolic execution and program testing. *Communications of the ACM* 19(7): 385–394.

Klocwork website. http://www.klocwork.com/

Korel, B. 1990. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8): 870–879.

LLVM website. http://llvm.org/

Lutz, R.R., G.C. Helmer, M.M. Moseman, and D.E Statezni. 1998. Safety analysis of requirements for a product family. In *Proceedings of the Third IEEE International Conference on Requirements Engineering* (ICRE'98), pp. 24–33.

MAGIC: Modular Analysis of proGrams In C. http://www.cs.cmu.edu/~chaki/magic/

Manna, Z. and A. Pnueli. 1990. Tools and rules for the practicing verifier. Technical Report STAN-CS-90-1321, Department of Computer Science, Stanford University. Appeared in *Carnegie Mellon Computer Science: A 25 year Commemorative*.

Marinov, D., and S. Khurshid. 2001. TestEra: A novel framework for automated testing of Java programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering* (ASE).

Markosian, L., M. Mansouri-Samani, P. Mehlitz, T. Pressburger. 2007. Program model checking using Design-for-Verification: NASA flight software case study. In *Proceedings of the 2007 IEEE Aerospace Conference*.

Mehlitz, P.C. and J. Penix. 2005. Design for verification with dynamic assertions. In *Proceedings of the 2005 29th Annual IEEE/NASA Software Engineering Workshop* (SEW'05), pp. 285–292.

Menzies, T., D. Owen, M. Heimdahl, J. Gao, and B. Cukic. 2004. Nondeterminism: Unsafe? Submitted to *IEEE Transactions on Software Engineering*. http://www.cs.pdx.edu/~timm/lite/readings/randomoka.pdf

Meyer, B. 1992. Applying "design by contract." *IEEE Computer,* 25(10): 40–51.

Meyers, S. 2005. *Effective* C++, Third Edition. Boston, MA: Addison Wesley.

MISRA - The Motor Industry Software Reliability Association. 2004. *Guidelines for the Use of the C Language in Critical Systems*.

Modex website. http://cm.bell-labs.com/cm/cs/what/modex/

MPL: Mars Polar Lander Wikipedia entry. http://en.wikipedia.org/wiki/Mars_Polar_Lander

Musuvathi, M., D.Y.W. Park, A. Chou, D.R. Engler, and D.L. Dill. 2002. CMC: A pragmatic approach to model checking real code. In *OSDI 2002: Proceedings of the 5th Symposium on Operating systems design and implementation*, pp. 75–88.

NASA 2004. *NASA Software Safety Guidebook*. NASA-GB-8719.13, Section 5.6.5.

National Institute of Standards and Technology. 2002. The economic impacts of inadequate infrastructure for software testing. Planning report 02-3, May 2002.

Naumovich, G., G. S. Avrunin, and L. A. Clarke. 1999. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings, 21st International Conference on Software Engineering*, pp. 399–410.

Olender, K. M. and Leon J. Osterweil. 1990. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering*, 16(3): 268–280.

The Omega Project: http://www.cs.umd.edu/projects/omega/

Park, D.Y.W., U. Stern, J.U. Skakkebaek, and D.L. Dill. 2000. Java model checking. In *Automated Software Engineering, 2000. Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering*, pp. 253–256.

Pasareanu, C., and W. Visser. 2004. Verification of Java programs using symbolic execution and invariant generation. In *Proceedings of the the 11th International SPIN Workshop on Model Checking of Software*.

Pasareanu, C., R. Pelanek, and W. Visser. 2005. Concrete model checking with abstract matching and refinement. In *Proceedings of the 17th International Conference on Computer Aided Verification* (CAV'05), Edinburgh, Scotland, pp. 52–66.

Parasoft website. http://www.parasoft.com/jsp/home.jsp

Penix, J., W. Visser, E. Engstrom, A. Larson, and N.Weininger. 2000. Verification of time partitioning in the DEOS scheduler kernel. In *Proceedings of the 22nd International Conference on Software Engineering*, pp. 488–497.

Penix, J., W, Visser, S. Park, C. Pasareanu, E. Engstrom, A. Larson, and N.Weininger. 2005. Verifying time partitioning in the DEOS scheduling kernel. *Formal Methods in Systems Design Journal*, 26(2): 103–135.

Pnueli, A. 1984. In transition from global to modular temporal reasoning about programs. In *Logic and Models of Concurrent Systems*, volume 13, pp. 123–144.

Polyspace Technologies website. http://www.polyspace.com/

Queille, J.P. and J. Sifakis. 1982. Specification and verification of concurrent systems in CAESAR. In M. Dezani and U. Montanari, eds. *Proceedings of the 5th Colloquium of the International Symposium on Programming* (ISP'82). Lecture Notes in Computer Science, vol. 137:337–351. London, UK: Springer-Verlag.

RTCA, Inc. website. http://www.rtca.org/

Robby, M.B. Dwyer, and J. Hatcliff. 2003. Bogor: An extensible and highly modular model checking framework. In *Proceedings of the Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering* (ESEC/FSE 2003).

Robby, M.B. Dwyer, and J. Hatcliff. 2006. Bogor: A flexible framework for creating software model checkers. In *Proceedings of Testing: Academic & Industrial Conference – Practice And Research Techniques*, Technical Report, SAnToS-TR2006-2. http://bogor.projects.cis.ksu.edu/

Rushby, J. 1999. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347; also issued by the FAA.

Saïdi, H. 1999. Modular and incremental analysis of concurrent software systems. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, pp. 92–101.

Sen, K., and G. Agha. 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *Proceedings of the The International Conference on Computer Aided Verification* (CAV 2006), pp. 419–423.

SmartEiffel. The GNU Eiffel compiler (open source). http://smarteiffel.loria.fr/

The SMV System. http://www.cs.cmu.edu/~modelcheck/smv.html

SPARKAda. Praxis High Integrity Systems. http://www.praxis-his.com/sparkada/

Spec Patterns website. http://patterns.projects.cis.ksu.edu/

Stoller, S.D. 2000. Model-checking multi-threaded distributed Java programs. In *Proceedings of the 7th International SPIN Workshop on Model Checking of Software*, pp. 71–91. Lecture Notes in Computer Science 1885. Berlin: Springer-Verlag.

Sutter, H. 1999, *Exceptional C++*. Boston, MA: Addison-Wesley.

Sutter, H. 2004. *C++ Coding Standards*. Boston, MA: Addison-Wesley.

Thompson, S. and G. Brat. 2008. Verification of C++ Flight Software with the MCP Model Checker. In *Proceedings of the 2008 IEEE Aerospace Conference* (to appear)

Tkachuk, O., M. B. Dwyer, and C. Pasareanu. 2003. Automated environment generation for software model checking. In *Proceedings of the Eighteenth IEEE International Conference on Automated Software Engineering*, pp. 116–127.

Tonella, P. 2004. Evolutionary testing of classes. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2004), pages 119−128.

T-VEC website. http://www.t-vec.com

TVLA Logic Analysis Engine. http://www.cs.tau.ac.il/~tvla/

Verisoft website. http://cm.bell-labs.com/who/god/verisoft/

Visser, W., K. Havelund, G. Brat, S. Park, and F. Lerda. 2003. Model checking programs. Journal of Automated Software Engineering, 10(2): 203–232.

Visser, W., C. Pasareanu, and S. Khurshid. 2004. Test input generation in Java PathFinder. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2004), pp. 97–107.

Visser, W., C. Pasareanu, and R. Pelánek. 2006. Test input generation for Java containers using state matching. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2006), pp. 37–48.

Wolper, P. 1985. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 184–193.

Yang, C-S.D., A.L. Souter, and L.L. Pollock. 1998. All-du-path coverage for parallel programs. *ACM SIGSOFT Software Engineering Notes*, 23(2): 153–162.

Xie, T., D. Marinov, and D. Notkin. 2004. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering* (ASE).

Xie, T., D. Marinov, W. Schulte, and D. Notkin. 2005. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for Construction and Analysis of Systems* (TACAS'05).